Towards Automatic Inference of Inductive Invariants

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

Distributed systems are notoriously difficult to design and implement correctly. Formal verification provides correctness proofs, and has recently been successfully applied to various distributed systems. At the heart of a typical formal verification is a computer-checked proof with an *inductive invariant*. Finding this inductive invariant is the hardest part of the proof: a part that is currently undertaken manually by the developer and is responsible for most of the effort associated with formal verification.

In this paper, we present a new approach: Incremental Inference of Inductive Invariants (I4), to automatically generate inductive invariants for distributed protocols. We start from a simple idea: the inductive invariant of a *finite* instance of the protocol must be an instance of a general inductive invariant for the *infinite* distributed protocol. In I4, we instantiate a finite instance of the protocol, work out the finite inductive invariant of this instance, then figure out the general inductive invariant as a generalization of the finite invariant. Our experiments show that I4 can finish the general proof of correctness of several systems with minimal human effort.

ACM Reference Format:

Haojun Ma, Aman Goel, Jean-Baptiste Jeannin, Manos Kapritsos, Baris Kasikci, and Karem A. Sakallah. 2019. Towards Automatic Inference of Inductive Invariants. In *Workshop on Hot Topics in Operating Systems (HotOS '19), May 13–15, 2019, Bertinoro, Italy*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3317550.3321451

1 Introduction

For more than 50 years, the systems community and the industry have been relying on testing to increase their confidence in the correctness of software [3, 5, 17, 27, 39]. As the availability demands started to increase, however, we realized that perhaps testing is not enough: even the most thorough testing discipline is bound to occasionally miss a bug, which may manifest during production, resulting in loss of availability, revenue, and company reputation [10, 41, 42, 44]. This has led many researchers and companies to look for alternative ways to develop software with strong correctness guarantees.

Thankfully, our increasing need for availability has been paralleled by an increase in the capabilities of formal verification techniques. In the last five years, the systems community has embraced these tools and techniques enthusiastically and has started proposing solutions for building provably-correct systems [7, 8, 22, 23, 28, 34, 38].

However, existing approaches to formally verifying complex systems are fundamentally unscalable. They use interactive and automated theorem provers [11, 32, 35, 36, 40] to dispatch a number of proof obligations, thus making the proof easier. At the heart of every proof, however, lies a critical process that these approaches cannot automate: finding an *inductive invariant*. An inductive invariant is an invariant—i.e., a safety property that must always hold during an execution—that can be shown to hold inductively. In all but the simplest systems, the safety property that one sets out to prove—e.g., at most one node holds the lock at all times—is indeed an invariant; but this invariant is not strong enough to support an inductive argument—i.e., that if the invariant holds in some state and the system transitions to another state, the invariant will still hold. An inductive invariant is typically much stronger: it implies the desired safety property, but also enables inductive reasoning by including other clauses describing various aspects of the system.

Finding an inductive invariant is tantamount to proving the correctness of the system; but it is also the hardest part of the proof. Most importantly, these inductive invariants are very complicated, even for simple systems. As the system complexity increases, these invariants grow proportionally more complex, too. During the IronFleet project [22], 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. It is perhaps not surprising then, that when they attempted to

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
HotOS '19, May 13–15, 2019, Bertinoro, Italy
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6727-1/19/05…$15.00
https://doi.org/10.1145/3317550.3321451

find the inductive invariant of a real-world system—i.e. the Paxos protocol [29]—the required effort was in the order of months.

In this work, we explore an automated way of finding such inductive invariants; 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 we could thus infer them 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 idea of 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.

A crucial requirement of the I4 process is that we be able to automatically identify the inductive invariant for a small instance of the system. To that end, we leverage the decades of progress made by the model checking community. Model checking is typically considered inadequate [2, 22, 43] for proving the correctness of real-world distributed systems, as the state space it must explore increases rapidly. While this state 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 [4] has shown that it is possible, given a finite and moderately complex system instance, to either provide 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.

2 Related Work

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

2.1 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+ [30] 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 [22] and Verdi [43]. IronFleet uses a combination of refinement and reduction [33] 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 *manual effort* to identify the inductive invariants of the system—and thus prove its correctness.

Ivy [36] aims to reduce that effort by facilitating the hardest part about proving correctness properties: finding an inductive invariant. To achieve that, Ivy restricts the implementation enough to ensure that it includes no undecidable proposition. 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.

2.2 Inductive Invariants

To overcome the challenge of finding an inductive invariant, previous work has taken a number of approaches for both finite and infinite programs. Daikon [14] was proposed in 2000 to learn possible program invariants, followed by Houdini [15] which learns conjunctive inductive invariants. IC3 [4] and PDR [13] can automatically find inductive invariants for finite state machines, and were later extended to certain systems with infinite-domain variables [9, 24]. For list-manipulating programs, Property-Directed Shape Analysis [25] and UPDR [26] have shown effective results, though the approach doesn't guarantee termination. Grebenshchikov et al. [21] show that $P \wedge T \implies P'$ (i.e., a Horn clause) is the most common pattern in program verification, and the ICE learning model [12, 16] 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 infinite copies of a state variable in different nodes).

3 Algorithm

The notion that verifying a parameterized distributed system consisting of n identical interacting processes can be accomplished by verifying a relatively small finite instantiation was first proposed by Pnueli et al. [37] 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 fundamental idea is the inspiration behind I4. Instead of computing n_0, however, we perform verification on

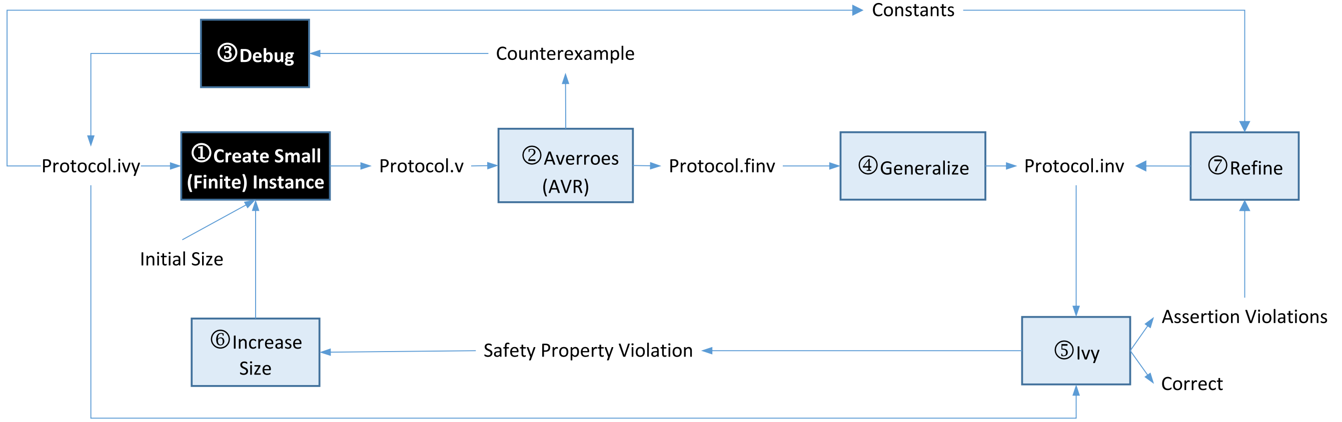

Figure 1. Flow of I4. Light blue boxes are fully automated, black boxes are partially automated.

instances of increasing size until we obtain the desired inductive invariant. This is done primarily because Pnueli et al.'s result does not readily apply to protocols where the state space in every process is infinite, which is a common occurrence in distributed systems. We show in Section 4 that even for infinite-state systems, I4 can infer the inductive invariant from a very small instance.

To model-check invariant instances on a small number of processes, we leverage model-checking techniques and our Averroes Formal Hardware Verification system [18, 31], which we are using as a proof-of-concept demonstration. Averroes v2.0 (AVR) [18] is a word-level IC3-based model checker that exploits data abstraction to significantly reduce verification complexity and has shown very good performance for hardware model checking [19, 20]. Given a finite transition system, a predicate on initial states, and a safety property that must hold on all reachable states, AVR will automatically produce either (i) an inductive invariant, i.e. a conjunction of the safety property and a number of *strengthening assertions*; or (ii) a counterexample showing how the property is violated.

Figure 1 shows the high-level design flow of the I4 system. Starting with a distributed protocol described in the Ivy language [36], we create an initial small finite instance of the protocol (step ①). In this step we bound everything to a small finite instance—e.g. the number of nodes in the protocol, the number of views in Paxos, etc. This is currently done manually, but the process is mostly mechanical and we aim to automate it in the future. In our prototype implementation, this finite instance is encoded in the SystemVerilog Hardware Description language [1]. This may seem like an odd choice for software verification. Indeed, our initial prototype uses Verilog out of mere convenience, as AVR natively supports it. Our end goal is to encode the protocol in a higher-level language, like Ivy [36] or Dafny [32], and then automatically translate that description into a format that AVR understands.

In step ②, when AVR is applied to a finite instance of a distributed protocol, it either produces a counterexample trace as evidence of how the protocol fails to satisfy the safety property or an inductive invariant specific to that finite instance (`protocol.finv`). Failure of the property on a finite instance of the protocol indicates the presence of bugs in the protocol or property specification, and we need to manually inspect the counterexample to debug the protocol (step ③).

Assuming a successful outcome from AVR, the next step in the I4 flow is to generalize the finite invariant to instances of arbitrary size in step ④ by universally quantifying the strengthening assertions produced by AVR. Consider, for example, the strengthening assertion $P(N_1)$, where P denotes an arbitrary predicate and N_1 is one of the nodes in the finite instantiation of the protocol. This assertion is generalized to apply for all nodes; i.e. $\forall N_1 .\, P(N_1)$. Assertions involving different nodes require a slightly more complex generalization. For example, an assertion such as $P(N_1) \wedge Q(N_2)$ where P and Q are different predicates is generalized to $\forall N_1, N_2 .\, (N_1 \neq N_2) \implies P(N_1) \wedge Q(N_2)$.

The generalized assertions are then combined with the original protocol and passed on to Ivy (step ⑤), which checks if they are sufficient to prove the correctness of the protocol. Ivy will check if the conjunction of all assertions is inductive. In particular, it tries to answer the following question: given a transition from state s to state s', such that the conjunction of all assertions holds for s, does each assertion hold for s'? This question is asked separately for each assertion. There are three possible outcomes:

1. The generalized assertions are sufficient 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 assertions was too small to capture all behaviors of the distributed protocol. In this case, we need to create an

instance with a larger size (with more nodes, etc.) and repeat the process (step ⑥).

3. Ivy fails to prove one or more of the generalized assertions on s'. There are two possible reasons for this failure. First, it is possible that some assertion A is too strong: even when assuming that the conjecture of all assertions holds for s, that is not enough to prove that A holds in s'. In this case, I4 uses an automatic refinement step to weaken the assertion (step ⑦).

   This step consists of using a number of heuristics to identify an appropriate weakening for the failing assertion. We consider, for example, whether the assertion holds only for "special" nodes (e.g. the node that initially holds the lock). Such special nodes are typically defined as constants in the protocol description. We use these constants as hints to perform the appropriate weakening. For example, when weakening the assertion $\forall N_1, N_2.\ (N_1 \neq N_2) \implies P(N_1) \wedge Q(N_2)$, we consider the fact that N_1 is actually the node that initially holds the lock—denoted as $N_1 = first$ in our instantiation. The weakened assertion then becomes $\forall N_1, N_2.\ (N_1 = first) \implies ((N_1 \neq N_2) \implies P(N_1) \wedge Q(N_2))$. After we refine all failing assertions in the inductive invariant, we feed it back to Ivy to check if further refinement is necessary. I4 automatically repeats this process until the proof succeeds or until our heuristics are exhausted and no refinement is possible.

Overall, the I4 methodology starts with a too-strong version of the invariant and applies heuristics to incrementally weaken assertions that are too strong. The caveat, of course, is that if our weakening is too aggressive, we will end up with a too-weak assertion. Currently we do not have a mechanism for identifying and strengthening too-weak assertions. If this happens, I4 will keep weakening the invariant until it is too weak to support the safety property, at which point we are back in case (2) and need to consider a larger instance.

4 Evaluation

We evaluate the ability of I4 to infer inductive invariants by testing it on three case studies: a **lock server**, a **leader election** algorithm, and a **distributed lock** protocol. For all three cases, we manually instantiated a finite instance of the protocol. From that instance, I4 was then able to automatically infer a general inductive invariant that passes Ivy's verification.

Before we discuss I4's success stories, however, we should caution the reader against generalizing these successes to larger and more complicated protocols. Our results thus far have been promising; but they do not yet scale to complex distributed protocols, like Paxos [29]. Section 5 discusses some of the limitations and future directions of this work.

4.1 Lock Server

Our first case study is a simple lock server[36, 43], a protocol with both infinite clients and infinite servers. Every server maintains a lock and the server's state is a *semaphore* indicating if it currently holds its lock. Every client-server pair is associated with a boolean *link(C,S)* which denotes if 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 and acquire that server's lock, if that server currently holds its lock. A client may also release a lock, handing it back to the server. In this protocol, the safety property is defined as "no two clients can have a link to (i.e. hold the lock of) the same server at the same time":

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

We instantiated this protocol with 1 server and 4 clients, and used AVR to obtain the inductive invariant of this particular instance. The inductive invariant is:

$$\begin{array}{|ll|}\hline \neg(semaphore_0 \wedge link_0_0) & \wedge \\ \neg(semaphore_0 \wedge link_1_0) & \wedge \\ \neg(semaphore_0 \wedge link_2_0) & \wedge \\ \neg(semaphore_0 \wedge link_3_0) & \wedge \\ Safety\ Property & \\ \hline \end{array}$$

It turns our that the above inductive invariant is the easiest to generalize. In fact I4 only needed to apply step ④: putting universal quantifiers before every strengthening assertion. The resulting generalized inductive invariant below indeed passes Ivy's verification without any manual effort.

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

This first result is promising, but as the protocols and their inductive invariants get more complicated, mere generalization will not be enough; I4's refinement process will have to come into the picture.

4.2 Leader Election

Our second case study is more complex: it is a protocol for leader election on a ring [6, 36]. This protocol has an unbounded number of nodes, but each node can only communicate with its two neighbors on the ring. Each node has a unique ID and the goal of the algorithm 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, it is clear that no other ID is larger than its own and it becomes the leader.

In this protocol, the safety property 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 involves at least three nodes, we use an instance with three nodes to generate the finite inductive invariant:

$$\begin{array}{|ll|}\hline
\neg((pending_0 = idn_0) \wedge \neg(idn_2 \leq idn_0)) & \wedge \\
\neg((pending_0 = idn_0) \wedge (idn_0 \leq idn_1)) & \wedge \\
\neg((pending_0 = idn_1) \wedge (idn_1 \leq idn_2)) & \wedge \\
\neg((pending_1 = idn_1) \wedge \neg(idn_0 \leq idn_1)) & \wedge \\
\neg((pending_1 = idn_1) \wedge (idn_1 \leq idn_2)) & \wedge \\
\neg((pending_1 = idn_2) \wedge (idn_2 \leq idn_0)) & \wedge \\
\neg((pending_2 = idn_0) \wedge (idn_0 \leq idn_1)) & \wedge \\
\neg((pending_2 = idn_2) \wedge \neg(idn_1 \leq idn_2)) & \wedge \\
\neg((pending_2 = idn_2) \wedge (idn_2 \leq idn_0)) & \wedge \\
\neg(leader_0 \wedge \neg(idn_2 \leq idn_0)) & \wedge \\
\neg(leader_0 \wedge (idn_0 \leq idn_1)) & \wedge \\
\neg(leader_1 \wedge \neg(idn_0 \leq idn_1)) & \wedge \\
\neg(leader_1 \wedge (idn_1 \leq idn_2)) & \wedge \\
\neg(leader_2 \wedge \neg(idn_1 \leq idn_2)) & \wedge \\
\neg(leader_2 \wedge (idn_2 \leq idn_0)) & \wedge \\
Safety\ Property & \\ \hline
\end{array}$$

In this case, generalization (step ④) is not enough. Take the third (highlighted) clause of the finite inductive invariant above, for example. After generalization, this clause becomes:

$$\forall N0, N1, N2.\ (N0 \neq N1) \wedge (N0 \neq N2) \wedge (N1 \neq N2) \implies$$
$$\neg((pending(N0) = idn(N1)) \wedge le(idn(N1), idn(N2)))$$

This clause is now too strong. This property doesn't actually hold *for all* triplets of nodes $N0, N1, N2$. In fact, this property only holds if $N1$ is between $N0$ and $N2$ on the ring. The automatic refinement process of I4 realizes (using Ivy's feedback) that this clause is too strong and weakens it using the *btw* (read "between") relation found in the original protocol. The resulting inductive invariant is shown in Table 1, with the relevant clause highlighted for reference. This inductive invariant passes Ivy's verification, thus proving the correctness of the protocol without manual assistance.

4.3 Distributed lock protocol

Our last experiment is a distributed lock protocol [22, 36]. This protocol also has an unbounded number of nodes that transfer ownership of a lock among them. The ownership of a lock is associated with an ever increasing epoch number, to allow detection of stale messages. Initially the lock is held by a designated node called *first*.

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 them a $transfer(E, N')$. When a node N' at epoch $ep(N')$ receives a $transfer$ message with epoch $E' > ep(N')$, node N' accepts the lock at epoch E', and sends a *locked* message with epoch E' to denote that N' holds the lock at epoch E'. Otherwise, if $E' \leq ep(N')$, N' ignores this stale message. As the network may delay or duplicate any message, we maintain all possible $transfer(E, N)$ messages.

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

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

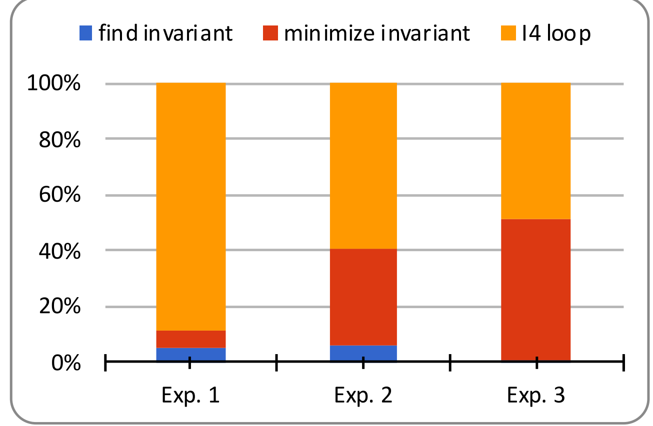

Figure 2. Runtime Break Down

This protocol involves two sources of infinity: the number of nodes and the number of epochs. Unlike previous examples, even an instance with just two nodes would be enough to generate an unbounded number of messages by passing the lock between them with ever increasing epoch numbers. We therefore need to bound not just the number of nodes, but also the number of epochs to derive a finite instance of the protocol. In fact, we were able to prove the correctness of the protocol based on a finite instance with just two nodes and four epochs.

Given the inductive invariant for this finite instance, I4 finds the general inductive invariant after seven iterations of refinement.

Scalability. In an attempt to understand the scalability of the I4 approach, we applied it to larger instances of the distributed lock protocol. Table 2 shows the evaluation of three instances of size $(2, 4)$, $(2, 8)$, and $(8, 4)$, where (m, n) indicates an instance with m nodes and n epochs. We find that both the complexity of the finite inductive invariant and the I4 execution time grow exponentially with the size of the system.

Figure 2 shows a breakdown of the I4 execution time. A large fraction of the time ($6\% \sim 50\%$) is currently taken by our invariant minimization process: we not only ask AVR for a finite inductive invariant, but we further ask it to remove any redundancy from the invariant's clauses, if possible. This process produces more compact and readable invariants, but it is not really necessary for automation, so we aim to dispense with it in the future.

4.4 Comparison With Other Tools

The lock server protocol is taken from [36, 43]. With the Coq and Verdi framework, the authors write a proof that is approximately 500 lines long. In Ivy, it still requires 8 iterations of user-guided generalizations. I4 finishes the proof

Table 1. General Invariant of Leader Election

$\forall N0, N2. \ (N0 \neq N2) \implies \neg((pending(N0) = idn(N0)) \wedge \neg le(idn(N2), idn(N0))) \quad \wedge$
$\forall N0, N1, N2. \ ring.btw(N0, N1, N2) \wedge ring.btw(N1, N2, N0) \wedge ring.btw(N2, N0, N1) \implies$
$((N0 \neq N1) \wedge (N0 \neq N2) \wedge (N1 \neq N2) \implies \neg((pending(N0) = idn(N1)) \wedge le(idn(N1), idn(N2)))) \quad \wedge$
$\forall N0, N2. \ (N0 \neq N2) \implies \neg(leader(N0) \wedge \neg le(idn(N2), idn(N0))) \quad \wedge$
$Safety\ Property$

Table 2. Scalability results

	Exp.1	Exp.2	Exp.3
# of nodes	2	2	8
# of epochs	4	8	4
size of initial invariant	40	457	686
size of minimized invariant	38	209	602
# of refine iterations	8	15	14
total time(s)	11.960	378.306	5271.316

with automatic inductive invariant inference at our first attempt, in a few seconds.

The distributed lock protocol is taken from [22, 36]. With the Dafny and IronFleet framework, proving the correctness of this protocol took the authors several days, most of which was spent on identifying the inductive invariant. With Ivy, it still requires several hours to iterate with the user-guided generalizations. I4 proved the correctness of the protocol automatically and took only 12 seconds.

5 Discussion

Our experiments show that our algorithm can automate the process of finding inductive invariants, but there is much room for improvement:

- **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.
- **Instantiating.** Currently, we need to manually instantiate the protocol to a finite system and feed it to AVR. We plan to fully automate this process.
- **Leveraging Ivy's feedback.** We are currently using Ivy as a means to detect which strengthening assertions are part of an inductive invariant. We have plans to further leverage the counterexamples produced by Ivy to guide our refinement.
- **Heuristic refinement.** Heuristic refinement is critical in our algorithm. We are currently using a small number of heuristics during our refinement step. These heuristics are sufficient to perform refinement for the simple protocols we have considered so far. As we apply I4 to more complex protocols, we expect our arsenal of heuristics to grow.
- **Verifying implementation.** Our current approach 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.

6 Conclusion

We have presented I4, a tool for incremental inference of inductive invariants. I4 is based on a simple intuition: an inductive invariant of a small finite instance can be used to infer a general inductive invariant. We leverage the power of model-checking to automatically identify the invariants of small instances and we use an iterative refinement process to identify an invariant that holds for all instances of the system. I4 was successful in automatically inferring the inductive invariants for three simple distributed protocols. Our next steps will be to scale our approach to larger and more complex distributed systems.

References

[1] IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language. *IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012)*, pages 1–1315, Feb 2018.

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

[3] 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.

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

[5] 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.

[6] 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.

[7] 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.

[8] 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.

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

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

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

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *PLDI' 05*, pages 213–223, 2005.

[18] A. Goel and K. Sakallah. Averroes 2. http://www.github.com/aman-goel/avr.

[19] 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.

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

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

[22] 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.

[23] 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.

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

[25] 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.

[26] 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.

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

[28] 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.

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

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

[31] 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.

[32] 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.

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

[34] 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.

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

[36] 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.

[37] 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.

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

[39] 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.

[40] 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.

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

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

[43] 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. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 357–368, New York, NY, USA, 2015. ACM.

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