Formal Methods in Computer-Aided Design 2022

Plain and Simple Inductive Invariant Inference for Distributed Protocols in TLA+

William Schultz
Northeastern University
Boston, MA
schultz.w@northeastern.edu

Ian Dardik
Carnegie Mellon University
Pittsburgh, PA
idardik@andrew.cmu.edu

Stavros Tripakis
Northeastern University
Boston, MA
stavros@northeastern.edu

***Abstract*—We present a new technique for automatically inferring inductive invariants of parameterized distributed protocols specified in TLA+. Ours is the first such invariant inference technique to work directly on TLA+, an expressive, high level specification language. To achieve this, we present a new algorithm for invariant inference that is based around a core procedure for generating *plain*, potentially non-inductive *lemma invariants* that are used as candidate conjuncts of an overall inductive invariant. We couple this with a greedy lemma invariant selection procedure that selects lemmas that eliminate the largest number of counterexamples to induction at each round of our inference procedure. We have implemented our algorithm in a tool, *endive*, and evaluate it on a diverse set of distributed protocol benchmarks, demonstrating competitive performance and ability to uniquely solve an industrial scale reconfiguration protocol.**

I. Introduction

Automatically verifying the safety of distributed systems remains an important and difficult challenge. Distributed protocols such as Paxos [32] and Raft [39] serve as the foundation of modern fault tolerant systems, making the correctness of these protocols critical to the reliability of large scale database, cloud computing, and other decentralized systems [47], [8], [11], [38]. An effective approach for reasoning about the correctness of these protocols involves specifying system *invariants*, which are assertions that must hold in every reachable system state. Thus, a primary task of verification is proving that a candidate invariant holds in every reachable state of a given system. For adequately small, finite state systems, symbolic or explicit state model checking techniques [12], [26], [6] can be sufficient to automatically prove invariants. For verification of infinite state or *parameterized* protocols, however, model checking techniques may, in general, be incomplete [7]. Thus, the standard technique for proving that such a system satisfies a given invariant is to discover an *inductive invariant*, which is an invariant that is typically stronger than the desired system invariant, and is preserved by all protocol transitions. Discovering inductive invariants, however, is one of the most challenging aspects of verification and remains a non-trivial task with a large amount of human effort required [50], [13], [49], [44]. Thus, automating the inference of these invariants is a desirable goal.

This work was supported by the U.S. National Science Foundation under NSF SaTC award CNS-1801546.

In general, the problem of inferring inductive invariants for infinite state protocols is undecidable [40]. Even the verification of inductive invariants may require checking the validity of arbitrary first order formulas, which is undecidable [41]. Thus, this places fundamental limits on the development of fully general algorithmic techniques for discovering inductive invariants.

Significant progress towards automation of inductive invariant discovery for infinite state protocols has been made with the Ivy framework [42]. Ivy utilizes a restricted system modeling language that allows for efficient checking of verification goals via an SMT solver such as Z3 [17]. In particular, the EPR and extended EPR subsets of Ivy are decidable. Ivy also provides an interface for an interactive, counterexample guided invariant discovery process. The Ivy language, however, may place an additional burden on users when protocols or their invariants don't fall naturally into one of the decidable fragments of Ivy. Transforming a protocol into such a fragment is a manual and nontrivial task [41].

Subsequent work has attempted to fully automate the discovery of inductive invariants for distributed protocols. State of the art tools for inductive invariant inference for distributed protocols include I4 [35], fol-ic3 [29], IC3PO [24], SWISS [25], and DistAI [51]. All of these tools, however, accept only Ivy or an Ivy-like language [2] as input. Moreover, several of these tools work only within the restricted decidable fragments of Ivy.

In this paper, we present a new technique for automatic discovery of inductive invariants for protocols specified in TLA+, a high level, expressive specification language [33]. To our knowledge, this is the first inductive invariant discovery tool for distributed protocols in a language other than Ivy. Our technique is built around a core procedure for generating small, *plain* (potentially non-inductive) invariants. We search for these invariants on finite protocol instances, employing the so-called *small scope* hypothesis [27], [35], [4], circumventing undecidability concerns when reasoning over unbounded domains. We couple this invariant generation procedure with an invariant selection procedure based on a greedy counterexample elimination heuristic in order to incrementally construct an overall inductive invariant. By restricting our inference reasoning to finite instances, we avoid restrictions imposed by modeling approaches that try to maintain decidability of

https://doi.org/

SMT queries.

Our technique is partially inspired by prior observations [13], [44], [25], [10] that, for many practical protocols, an inductive invariant I is typically of the form $I = P \wedge A_1 \wedge \cdots \wedge A_n$, where P is the main invariant (i.e. safety property) we are trying to establish, and $A_1, \ldots, A_n$ are a list of *lemma invariants*. Each lemma invariant A_i may not necessarily be inductive, but it is necessarily an invariant, and it is typically much smaller than I. These lemma invariants serve to strengthen P so as to make it inductive. Many prior approaches to inductive invariant inference have focused on searching for lemma invariants that are inductive, or inductive relative to previously discovered information [25], [10], [24], [29]. In contrast, our inference procedure searches for *plain* lemma invariants and uses them as candidates for conjuncts of an overall inductive invariant. To search for lemma invariants, we sample candidates using a syntax-guided approach [20], and verify the candidates using an off the shelf model checker.

We have implemented our invariant inference procedure in a tool, *endive*, and we evaluate its performance on a set of diverse protocol benchmarks, including 29 of the benchmarks reported in [24]. Our tool solves nearly all of these benchmarks, and compares favorably with other state of the art tools, despite the fact that all of these tools accept Ivy or decidable Ivy fragments as inputs. We also evaluate our tool and other state of the art tools on a more complex, industrial scale protocol, *MongoLoglessDynamicRaft (MLDR)* [44]. MLDR performs dynamic reconfiguration in a Raft based replication system. Our tool is the only one which manages to find a correct inductive invariant for MLDR.

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

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

The rest of this paper is organized as follows. Section II presents preliminaries and a formal problem statement. Section III describes our algorithm for inductive invariant inference, along with more details on our technique. Section IV provides an experimental evaluation of our algorithm, as implemented in our tool, *endive*. Section V examines related work, and Section VI presents conclusions and goals for future work.

II. Preliminaries and Problem Statement

1) TLA+: Throughout the rest of this paper, we adopt the notation of TLA+ [33] for formally specifying systems and their correctness properties. TLA+ is an expressive, high level specification language for specifying distributed and concurrent protocols. It has also been used effectively in industry for specifying and verifying correctness of protocol designs [5], [38]. Note that our tool accepts models written in TLA+. Figure 1 describes a simple lock server protocol [42], [49] in TLA+ which we will use as a running example.

2) Symbolic Transition Systems: The protocols considered in this paper can be modeled as parameterized *symbolic transition systems* (STSs), like the one shown in Figure 1. This STS is parameterized by two *sorts*, called *Server* and *Client* (Line 1). Each sort represents an uninterpreted constant symbol that can be interpreted as any set of values. In this paper we assume that sorts may only be interpreted over finite domains of distinct values e.g. $\mathit{Server} = \{a_1, \ldots, a_k\}$ and $\mathit{Client} = \{c_1, \ldots, c_k\}$.

In addition to types, a STS also has a set of *state variables*. A *state* is an assignment of values to all state variables. We use the notation $s \models P$ to denote that state s *satisfies* state predicate P, i.e., that P evaluates to true once we replace all state variables in P by their values as given by s.

The STS of Figure 1 has two state variables, called *locked* and *held* (Line 2). The state predicate *Init* specifies the possible values of the state variables at an *initial state* of the system (Lines 3-5). *Init* states that initially $\mathit{locked}[i]$ is TRUE for all $i \in \mathit{Server}$, and that $\mathit{held}[i]$ is $\{\}$ (the empty set) for all $i \in \mathit{Client}$. The predicate *Next* defines the *transition relation* of the STS (Lines 14-16). In TLA+, *Next* is typically written as a disjunction of *actions* i.e., possible symbolic transitions. In the example of Figure 1 there are two possible symbolic transitions: either some client c and some server s engage in a "connect" action defined by the *Connect*(c, s) predicate, or some client c and some server s engage in a "disconnect" action defined by the *Disconnect*(c, s) predicate.

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

The entire set of behaviors of the system is defined as a single temporal logic formula *Spec* (Line 17). In TLA+, *Spec* is typically defined as the TLA+ formula $\mathit{Init} \wedge \Box[\mathit{Next}]_{\mathit{Vars}}$, where $\Box$ is the "always" operator of linear temporal logic, and $[\mathit{Next}]_{\mathit{Vars}}$ represents a transition which either satisfies Next or is a *stuttering* step, i.e., where all state variables in Vars remain unchanged.

3) Invariants: In this paper we are interested in the verification of safety properties, and in particular *invariants*, which are state predicates that hold at all reachable states. Formally, a state predicate P is an *invariant* if $s \models P$ holds for every reachable state s. The model of Figure 1 contains one such candidate invariant, specified by the predicate *Safe* (Line 18). *Safe* states that there cannot be two different clients ci and cj which both hold locks to the same server.

$$
\begin{array}{l}
\textsc{constant}\ \mathit{Server},\ \mathit{Client} \\
\textsc{variable}\ \mathit{locked},\ \mathit{held} \\[4pt]
\mathit{Init}\ \triangleq \\
\qquad \land\ \mathit{locked} = [i \in \mathit{Server} \mapsto \textsc{true}] \\
\qquad \land\ \mathit{held} = [i \in \mathit{Client} \mapsto \{\}] \\[4pt]
\mathit{Connect}(c,\ s)\ \triangleq \\
\qquad \land\ \mathit{locked}[s] = \textsc{true} \\
\qquad \land\ \mathit{held}' = [\mathit{held}\ \textsc{except}\ ![c] = \mathit{held}[c] \cup \{s\}] \\
\qquad \land\ \mathit{locked}' = [\mathit{locked}\ \textsc{except}\ ![s] = \textsc{false}] \\[4pt]
\mathit{Disconnect}(c,\ s)\ \triangleq \\
\qquad \land\ s \in \mathit{held}[c] \\
\qquad \land\ \mathit{held}' = [\mathit{held}\ \textsc{except}\ ![c] = \mathit{held}[c] \setminus \{s\}] \\
\qquad \land\ \mathit{locked}' = [\mathit{locked}\ \textsc{except}\ ![s] = \textsc{true}] \\[4pt]
\mathit{Next}\ \triangleq \\
\qquad \lor\ \exists\, c \in \mathit{Client},\ s \in \mathit{Server} : \mathit{Connect}(c,\ s) \\
\qquad \lor\ \exists\, c \in \mathit{Client},\ s \in \mathit{Server} : \mathit{Disconnect}(c,\ s) \\[4pt]
\mathit{Spec}\ \triangleq\ \mathit{Init} \land \Box[\mathit{Next}]_{\langle \mathit{locked},\ \mathit{held} \rangle} \\[4pt]
\mathit{Safe}\ \triangleq \\
\qquad \forall\, c_i,\ c_j \in \mathit{Client} : \\
\qquad\quad (\mathit{held}[c_i] \cap \mathit{held}[c_j] \neq \{\}) \Rightarrow (c_i = c_j)
\end{array}
$$

Fig. 1. A simple parameterized protocol defined in TLA⁺.

4) Verification: The verification problem consists in checking that a system satisfies its specification. In TLA⁺, both the system and the specification are written as temporal logic formulas. Therefore, expressed in TLA⁺, the safety verification problem we consider in this paper consists of checking that the temporal logic formula

$$\mathit{Spec} \Rightarrow \Box \mathit{Safe} \tag{1}$$

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

5) Finite State Instances: Instantiating a sort means fixing it to a *finite* domain of distinct elements. For example, we can instantiate *Server* to be the set $\{a_1, a_2\}$ (meaning there are only two servers, denoted a_1 and a_2), and *Client* to be the set $\{c_1, c_2\}$ (meaning there are only two clients, denoted c_1 and c_2). For the parameterized symbolic transition systems considered in this paper, when we instantiate all sorts of an STS, the system becomes finite-state, i.e., the set of all possible system states is finite.

6) Inductive Invariants: A standard technique for solving the safety verification problem (1) is to come up with an *inductive invariant* [36]. That is, a state predicate Ind which satisfies the following conditions:

$$\mathit{Init} \Rightarrow \mathit{Ind} \tag{2}$$

$$\mathit{Ind} \land \mathit{Next} \Rightarrow \mathit{Ind}' \tag{3}$$

$$\mathit{Ind} \Rightarrow \mathit{Safe} \tag{4}$$

where Ind' denotes the predicate Ind where state variables are replaced by their primed, next-state versions. Conditions (2) and (3) are, respectively, referred to as *initiation* and *consecution*. Condition (2) states that Ind holds at all initial states.

$$A_1\ \triangleq\ \forall\, s \in \mathit{Server} : \forall\, c \in \mathit{Client} : \mathit{locked}[s] \Rightarrow (s \notin \mathit{held}[c]))$$

$$\mathit{Ind}\ \triangleq\ \mathit{Safe} \land A_1$$

Fig. 2. A lemma invariant, A_1, and an inductive invariant, Ind, for the protocol and safety property given in Figure 1.

Condition (3) states that Ind is *inductive*, i.e., if it holds at some state s then it also holds at any successor of s. Together these two conditions imply that Ind is also an invariant, i.e., that it holds at all reachable states. Condition (4) states that Ind is stronger than the invariant Safe that we are trying to prove. Therefore, if all reachable states satisfy Ind, they also satisfy Safe, which establishes (1). The difficulty is in coming up with an inductive invariant which satisfies the above conditions. The problem we consider in this paper is to infer such an inductive invariant automatically.

7) Lemma Invariants: An inductive invariant Ind typically has the form $\mathit{Ind}\ \triangleq\ \mathit{Safe} \land A_1 \land \cdots \land A_k$, where the conjuncts $A_1, ..., A_k$ are state predicates and we refer to them as *lemma invariants*. Observe that each A_i must itself be an invariant. The reason is that Ind must be an invariant, i.e., must contain all reachable states, and since Ind is stronger than (i.e., contained in) each A_i, each A_i must itself contain all reachable states. Furthermore, although all lemma invariants must be invariants, they need not be individually inductive. However, the conjunction of all lemma invariants together with the safety property Safe must be inductive. Figure 2 provides an example of an inductive invariant, Ind, for the protocol and safety property given in Figure 1. Ind contains a single lemma invariant, A_1.

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

III. Our Approach

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

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

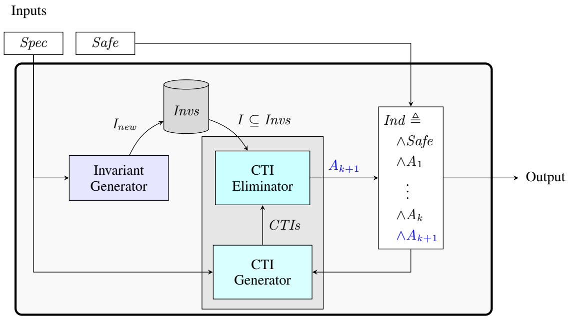

Fig. 3. Components of our technique for inductive invariant inference.

Algorithm 1 Our inductive invariant inference algorithm.

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

lemma invariants. Otherwise, add the selected lemma invariants to Ind and repeat from Step 2.

The conceptual approach is illustrated in Figure 3. Our detailed algorithm is described in Section III-A. Section III-B provides details on our lemma invariant generation procedure, Section III-C provides details on CTI generation, and Section III-D describes the selection of lemma invariants.

A. Inductive Invariant Inference Algorithm

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

The algorithm maintains a current inductive invariant candidate, Ind, which it initializes to $Safe$, the safety property that we are trying to prove (Line 3). It then generates a set X of CTIs of Ind (Line 4). The algorithm may also initialize the repository of lemma invariants, $Invs$, or add more lemma invariants to $Invs$ if it is initially non-empty (Line 5). The procedures $GenerateLemmaInvariants$ and $GenerateCTIs$ are described in more detail below, in Sections III-B and III-C, respectively.

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

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

If all known CTIs have been eliminated, the algorithm terminates successfully and returns Ind (Line 17). Successful termination of the algorithm indicates that the returned Ind is likely to be inductive. However our method does not provide a formal inductiveness guarantee. Ind might not be inductive for a number of reasons. First, as we discuss further in Section III-C, our CTI generation procedure is probabilistic in nature, and therefore $GenerateCTIs$ might miss some CTIs. Second, even if the finite-state instance M explored by the algorithm has no remaining CTIs, there might still exist CTIs in other instances of the STS , for larger parameter values.

Even though a candidate invariant returned by a successful termination of Algorithm 1 is not formally guaranteed to be inductive, we ensure soundness of our overall procedure by doing a final check that the discovered candidate inductive invariant is correct using the TLA+ proof system (TLAPS) [16]. Validation of invariants in TLAPS is discussed further in Section III-E. In practice we found that all of the invariants generated in our evaluation (Section IV) are correct inductive invariants.

We also remark that in the current version of our algorithm and in the current implementation of our tool, we only explore the single finite-state instance of the STS provided by the user, and we do not attempt to automatically increase the bounds of the parameters within the algorithm, as is done for example in the approach described in [24]. This is, however, a relatively straightforward extension to our algorithm, and would like to explore this option in future work.

B. Lemma Invariant Generation

For a given finite instance M of a parameterized transition system, the goal of lemma invariant generation is to produce

$$\langle seed \rangle ::= \; locked[s] \mid s \in held[c] \mid held[c] = \emptyset$$
$$\langle quant \rangle ::= \forall s \in Server : \forall c \in Client$$
$$\langle expr \rangle ::= \langle seed \rangle \mid \neg \langle expr \rangle \mid \langle expr \rangle \vee \langle expr \rangle$$
$$\langle pred \rangle ::= \langle quant \rangle : \langle expr \rangle$$

Fig. 4. Example of a grammar for lemma invariant generation for the *lockserver* protocol shown in Figure 1. The list of unquantified *seed* predicates and the quantifier template, *quant*, are provided as user inputs.

a set of state predicates that are invariants of M. To search for these invariants, we adopt an approach similar to other, *syntax-guided synthesis* based techniques [21], [20] for invariant discovery. We randomly sample invariant candidates from a defined *grammar*, which is generated from a given set of $seed$ predicates. Each seed predicate is an atomic boolean predicate over the state variables of the system. Note that the parameterized distributed protocols that we consider in this paper typically have inductive invariants that are universally or existentially quantified over the parameters of the protocol or other values of the system state. So, our invariant generation technique assumes a fixed quantifier template that is provided as input. The provided seed predicates are unquantified predicates that can contain bound variables that appear in the given quantifier template. An example of a simple grammar for the protocol of Figure 1 is shown in Figure 4.

Candidate invariants are produced by generating random predicates over the space of seed predicates. Specifically, a candidate predicate is formed as a random disjunction of seed predicates, where each disjunct may be negated with probability $\frac{1}{2}$. The logical connectives $\{\vee, \neg\}$ are functionally complete [48], so they serve as a simple basis for generating candidate invariants, which we chose to reduce the invariant search space.

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

(1) Generate $Reach(M)$, the set of reachable states of M.
(2) Check that $s \models P$ for each predicate $P \in C$ and each $s \in Reach(M)$.

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

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

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

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

C. CTI Generation

Each round of our algorithm relies on access to a set of multiple CTIs, as a means to prioritize between different choices of new lemma invariants. To generate these CTIs, we use a probabilistic technique proposed in [34] that utilizes the TLC explicit state model checker [52]. Given a finite instance of a STS M with system states S, transition relation predicate $Next$, and given candidate inductive invariant Ind, the procedure $GenerateCTIs(M, Ind)$ works by calling the TLC model checker. TLC attempts to randomly sample states $s_0 \in S$ for which there exists a sequence of states $s_1, s_2, \ldots, s_{k-1}, s_k \in S$, such that both of the following hold:

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

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

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

In practice, during our evaluation we found that TLC was able to effectively generate many thousands of CTIs at each round of the inference algorithm using the above technique. This provided an adequately diverse distribution of CTIs for effectively guiding our counterexample elimination procedure, which we describe in more detail in Section III-D. Section IV presents more detailed metrics on CTI generation as measured when testing our implementation on a variety of protocol benchmarks. In future we feel it would be valuable to explore and compare with other, SMT/SAT based techniques for this type of counterexample generation task [18], [30].

D. Lemma Invariant Selection by CTI Elimination

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

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

More generally, lemma selection at each round of the algorithm can be viewed as a version of the set covering problem [15]. Ideally, we would like to find the smallest set of lemma invariants that eliminate (i.e. cover) the set of CTIs X. Solving this problem optimally is known to be NP-complete [28], but we have found a greedy heuristic [14] to work sufficiently well in our experiments, the results of which are presented in Section IV. In future we would like to explore more sophisticated heuristics for lemma selection that take into account additional metrics, like syntactic invariant size, quantifier depth, etc.

E. Validation of Inductive Invariant Candidates

If our inference algorithm terminates successfully, it will return a candidate inductive invariant. Since we look for invariants on finite protocol instances, though, this candidate may not be an inductive invariant for general (e.g. unbounded) protocol instances. So, upon termination, we check to see if the returned candidate invariant is truly inductive for all protocol instances by passing it to an SMT solver. Currently, we use the TLA+ proof system (TLAPS) [16] for this step, which generates an SMT encoding for TLA+ [37].

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

IV. Implementation and Evaluation

A. Implementation and Experimental Setup

Our invariant inference algorithm is implemented in a tool, *endive*, whose main implementation consists of approximately 2200 lines of Python code. There are also some optimized subroutines which consist of an additional few hundred lines of C++ code. Internally, *endive* makes use of version 2.15 of the TLC model checker [52], with some minor modifications to improve the efficiency of checking many invariants simultaneously. TLC is used by *endive* for most of the algorithm's compute intensive verification tasks, like checking candidate lemma invariants (Section III-B) and CTI elimination checking (Section III-D1).

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

For each benchmark run, we initialize Invs (as explained in Algorithm 1) as an empty set and configure the lemma invariant generation procedure discussed in Section III-B with a parameter value of $N_{lemmas} = 15000$. The grammars used for invariant generation were mined from predicates appearing in each protocol specification.

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

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

B. Benchmarks

To evaluate *endive*, we measured its performance on 29 protocols selected from an existing benchmark set published in [24]. We also evaluate endive on an additional, industrial scale protocol, *MongoLoglessDynamicRaft (MLDR)*, which is a recent protocol for distributed dynamic reconfiguration in a Raft based replication system [45], [44].

1) Protocol Conversion: The 29 benchmarks we used from [24] were originally specified in Ivy [42], but *endive* accepts protocols in TLA+, so it was necessary to manually translate the protocols from Ivy to TLA+. There are significant differences in how protocols are specified in Ivy and TLA+. The underlying approach to modeling systems as discrete transition systems, however, by specifying initial states and a transition relation, are common between them. In our manual translation, we aimed to emulate the original Ivy model as close as possible.

The formal specification for the *MongoLoglessDynamicRaft* protocol (*MLDR*) was originally written in TLA+ [45]. Thus, in order to compare with other invariant inference tools which accept Ivy as their input language, we had to translate MLDR from TLA+ into Ivy. This conversion process was highly nontrivial due to the significant differences between the Ivy and TLA+ languages. TLA+ is a very expressive language that includes integers, strings, sets, functions, records, and sequences as primitive data types along with their standard semantics. In contrast, the Ivy modeling language, RML [42], includes only basic, first order relations and functions. For more complex datatypes (e.g. arrays or sequences), their semantics must be defined and axiomatized manually.

An artifact containing all of our source code and instructions for reproducing our evaluation results can be found at [43]. A public, open-source version of our tool is also available at [1].

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

TABLE I
DISTRIBUTED PROTOCOL BENCHMARK RESULTS.

C. Results

Our overall results are shown in Table I. We compared *endive* with four recent, state of the art techniques for inferring invariants of distributed protocols: IC3PO [24], fol-ic3 [29], SWISS [25], and DistAI [51]. Note that *endive* accepts protocols in TLA+, whereas all other tools accept protocols in Ivy or mypyvy.

The numbers shown for both IC3PO and fol-ic3 in Table I are as reported in the evaluation presented in [24], with timeouts indicated by a *TO* entry. For the SWISS results in Table I, where possible, we show the runtime numbers reported in [25], indicated with a † mark. For the benchmarks in Table I that were not tested in [25], we present the results from our own runs of the tool, all using default SWISS configuration parameters. We ran SWISS both with an invariant template matching our own template for *endive* and also in automatic mode, and report the better of the two results. The results for DistAI are reported from our runs using the tool in its default configuration. For DistAI and SWISS, we report an *err* result in cases where the tool returned an error without producing a result. We report a *fail* result in cases where DistAI or SWISS terminated without error but did not discover an inductive invariant. In all cases where a benchmark protocol was not available in the required input language for the corresponding tool, we mark this with an *n/a* entry.

For each benchmark result in Table I, we report the total wall clock time to discover an inductive invariant in the *Time* column, along with the number of total lemma invariants contained in the discovered invariant, including the safety property, in the *Inv* column. Note that the number of total lemmas in the invariants discovered by SWISS was not reported in [25]. Thus, we report the number of lemmas discovered by SWISS in our own runs, for the cases where we were able to run SWISS successfully to produce an invariant.

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

D. Comparison with Other Tools

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

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

E. Discussion

Our tool, *endive*, was able to successfully discover an inductive invariant for 25 of the 29 protocol benchmarks from [24], and all of the invariants it discovered were proven correct using TLAPS. For the two protocols out of these 29 that our tool did not solve, pyv-learning-switch and i4-learning-switch, this was due to scalability limitations of CTI generation, which we believe could be improved with a smarter CTI generation algorithm or by incorporating a symbolic model checker [30] for this task.

endive was also able to automatically discover an inductive invariant for a key safety property of *MLDR*, a Raft-based distributed dynamic reconfiguration protocol [45]. This protocol, reported in Table I as *mldr*, is a significantly more complex, industrial scale protocol [44]. IC3PO was not able to discover an invariant for our Ivy model of the MLDR protocol after a 1 hour timeout when given the same instance size used in the TLA$^+$ model given to *endive*. SWISS and DistAI both produced an error when run on our Ivy model of MLDR.

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

It is additionally worth noting that our current *endive* implementation is not highly optimized. In particular, the TLC model checker, used internally by endive, is implemented in Java and interprets TLA$^+$ specifications dynamically [31], rather than compiling models to a low level, native representation as done by tools like SPIN [26]. As a result, TLC may not be the most efficient for our inference procedure, and could likely be optimized further.

V. Related Work

There are several recently published techniques that attempt to solve the problem of inductive invariant inference for distributed protocols. The IC3PO tool [24], which extended the earlier I4 tool [35], uses a technique based on IC3 [10] with a novel *symmetry boosting* technique that serves to accelerate IC3/PDR and also to infer the quantifier structure of lemma invariants. The fol-ic3 algorithm presented in [29] presents another IC3 based algorithm which uses a novel *separators* technique for discovering quantified formulas to separate positive and negative examples during invariant inference. SWISS [25] is another recent approach that uses an enumerative search for quantified invariants while using the Ivy tool to validate possible inductive candidates. It relies on SMT based reasoning over an unbounded domain, and does not reason directly about finite instances of distributed protocols. DistAI [51] uses a similar approach but additionally utilizes a technique of sampling reachable protocol states to filter invariants, which is similar to our approach of executing explicit state model checking as a means to quickly discover invariants. DistAI is limited, however, to learning only universally quantified invariants.

In addition to these inductive invariant inference techniques, there also exists prior work on alternative techniques for parameterized protocol verification. These include approaches based on cutoff detection [3], regular model checking [9], and symbolic backward reachability analysis [23].

More broadly, there exist many prior techniques for the automatic generation of program and protocol invariants that rely on data driven or grammar based approaches. Houdini [22] and Daikon [19] both use enumerative checking approaches to discover program invariants. FreqHorn [20] tries to discover quantified program invariants about arrays using an enumerative approach that discovers invariants in stages and also makes use of the program syntax. Other techniques have also tried to make invariant discovery more efficient by using improved search strategies based on MCMC sampling [46].

VI. Conclusions and Future Work

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

In future, our tool can be extended to allow for automatic quantifier template search, and further optimizations can be made to the lemma invariant generation and selection procedures. It would be interesting to explore ways in which the invariant generation procedure can be guided more directly by the generated counterexamples to induction, as a means to prune the search space of candidate invariants more efficiently, perhaps using techniques similar to those presented in [46]. We would also be interested to see if quantifier structures can be inferred from the protocol syntax itself. Improving the performance of TLC, or experimenting with other, more efficient model checkers [26] would be another avenue, since model checking performance is a main bottleneck of our current approach.

References

[1] endive invariant inference tool, Github repository. https://github.com/will62794/endive, 2022.

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

[3] Parosh Abdulla, Frédéric Haziza, and Lukáš Holík. Parameterized verification through view abstraction. *Int. J. Softw. Tools Technol. Transf.*, 18(5):495–516, Oct 2016.

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

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

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

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

[8] James Bornholt, Rajeev Joshi, Vytautas Astrauskas, Brendan Cully, Bernhard Kragl, Seth Markle, Kyle Sauri, Drew Schleit, Grant Slatton, Serdar Tasiran, Jacob Van Geffen, and Andrew Warfield. Using Lightweight Formal Methods to Validate a Key-Value Storage Node in Amazon S3. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 836–850. Association for Computing Machinery, 2021.

[9] Ahmed Bouajjani, Peter Habermehl, and Tomáš Vojnar. Abstract Regular Model Checking. In Rajeev Alur and Doron A. Peled, editors, *Computer Aided Verification*, pages 372–386, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

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

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

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

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

[14] V. Chvatal. A greedy heuristic for the set-covering problem. *Math. Oper. Res.*, 4(3):233–235, aug 1979.

[15] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009.

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

[17] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[18] Rafael Dutra, Kevin Laeufer, Jonathan Bachrach, and Koushik Sen. Efficient Sampling of SAT Solutions for Testing. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, page 549–559. Association for Computing Machinery, 2018.

[19] Michael D Ernst, Jeff H Perkins, Philip J Guo, Stephen McCamant, Carlos Pacheco, Matthew S Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Science of computer programming*, 69(1-3):35–45, 2007.

[20] Grigory Fedyukovich and Rastislav Bodík. Accelerating Syntax-Guided Invariant Synthesis. In Dirk Beyer and Marieke Huisman, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 251–269, Cham, 2018. Springer International Publishing.

[21] Grigory Fedyukovich, Sumanth Prabhu, Kumar Madhukar, and Aarti Gupta. Quantified Invariants via Syntax-Guided Synthesis. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification*, pages 259–277, Cham, 2019. Springer International Publishing.

[22] Cormac Flanagan and K. Rustan M. Leino. Houdini, an Annotation Assistant for ESC/Java. In *Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity*, FME '01, page 500–517, Berlin, Heidelberg, 2001. Springer-Verlag.

[23] Silvio Ghilardi and Silvio Ranise. MCMT: A Model Checker modulo Theories. In *Proceedings of the 5th International Conference on Automated Reasoning*, IJCAR'10, page 22–29, Berlin, Heidelberg, 2010. Springer-Verlag.

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

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

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

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

[28] Richard M. Karp. *Reducibility among Combinatorial Problems*, pages 85–103. Springer US, Boston, MA, 1972.

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

[30] Igor Konnov, Jure Kukovec, and Thanh-Hai Tran. TLA+ Model Checking Made Symbolic. *Proc. ACM Program. Lang.*, 3(OOPSLA), Oct 2019.

[31] Markus A Kuppe. A Verified and Scalable Hash Table for the TLC Model Checker: Towards an Order of Magnitude Speedup. Master's thesis, University of Hamburg., 2017.

[32] Leslie Lamport. The Part-Time Parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

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

[34] Leslie Lamport. Using TLC to Check Inductive Invariance. http://lamport.azurewebsites.net/tla/inductive-invariant.pdf, 2018.

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

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

[37] Stephan Merz and Hernán Vanzetto. Encoding TLA+ into Many-Sorted First-Order Logic. In Michael Butler, Klaus-Dieter Schewe, Atif Mashkoor, and Miklos Biro, editors, *Abstract State Machines, Alloy, B, TLA, VDM, and Z*, pages 54–69, Cham, 2016. Springer International Publishing.

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

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

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

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

[42] Oded Padon, Kenneth L McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. Ivy: Safety Verification by Interactive Generalization. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 614–630, 2016.

[43] William Schultz, Ian Dardik, and Stavros Tripakis. Artifact for FMCAD 2022 paper: Plain and Simple Inductive Invariant Inference for Distributed Protocols in TLA+. https://doi.org/10.5281/zenodo.6994922, August 2022.

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

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

[46] Rahul Sharma and Alex Aiken. From Invariant Checking to Invariant Inference Using Randomized Search. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*, pages 88–105. Springer, 2014.

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

[48] William Wernick. Complete sets of logical functions. *Transactions of the American Mathematical Society*, 51:117–132, 1942.

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

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

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

[52] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model Checking TLA+ Specifications. In Laurence Pierre and Thomas Kropf, editors, *Correct Hardware Design and Verification Methods*, pages 54–66, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

APPENDIX A
DETAILED BENCHMARK RESULTS

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

Recall that we limit the maximum number of generated CTIs to 10000 per round, as mentioned in Section IV-A. This explains why some protocol results for the *endive* tool report elimination of exactly 10000 CTIs. For example, for the tla-twophase benchmark, an inductive invariant was discovered in a single round of the algorithm loop (starting at Line 6 of Algorithm 1), so no more than 10000 CTIs were generated in the entire run. If the benchmark run eliminated greater than 10000 CTIs, this indicates that it ran for more more than 1 round.

Also, for protocols that eliminated 0 CTIs (e.g. tla-consensus, tla-tcommit), this indicates that the starting safety property was already inductive. Thus, no CTIs were ever generated and no lemma invariants were needed. Similarly, some protocols eliminated a nonzero amount of CTIs less than 10000 (e.g. ex-quorum-leader-election). This may be the case when no more than a single round of the algorithm was needed to discover an inductive invariant, or that the number of generated counterexamples at each round did not exceed 10000. Recall that, even within a single round of the algorithm, as shown in Algorithm 1, it is possible to discover multiple new lemma invariants.

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

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

TABLE II
DETAILED PROFILING RESULTS FOR THE *endive* RESULTS FROM TABLE I.

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

TABLE III
ADDITIONAL STATISTICS FOR *endive* RESULTS REPORTED IN TABLE I.

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