Simpler Specifications and Easier Proofs of Distributed Algorithms Using History Variables*

Saksham Chand and Yanhong A. Liu
Stony Brook University, Stony Brook, NY 11794
{schand,liu}@cs.stonybrook.edu

Abstract. This paper studies specifications and proofs of distributed algorithms when only message history variables are used, using Basic Paxos and Multi-Paxos for distributed consensus as precise case studies. We show that not using and maintaining other state variables yields simpler specifications that are more declarative and easier to understand. It also allows easier proofs to be developed by needing fewer invariants and facilitating proof derivations. Furthermore, the proofs are mechanically checked more efficiently.
We show that specifications in TLA$^+$ and proofs in TLA$^+$ Proof System (TLAPS) are reduced by 25% and 27%, repectively, for Basic Paxos, and 46% (from about 100 lines to about 50 lines) and 48% (from about 1000 lines to about 500 lines), respectively, for Multi-Paxos. Overall we need 54% fewer manually written invariants and our proofs have 46% fewer obligations. Our proof for Basic Paxos takes 26% less time than Lamport et al.'s for TLAPS to check, and our proofs for Multi-Paxos are checked by TLAPS within 1.5 minutes whereas prior proofs for Multi-Paxos fail to be checked in the new version of TLAPS.

1 Introduction

Reasoning about correctness of distributed algorithms is notoriously difficult due to a number of reasons including concurrency, asynchronous networks, unbounded delay, and arbitrary failures. Emerging technologies like autonomous cars are bringing vehicular clouds closer to reality [9], decentralized digital currencies are gathering more attention from academia and industry than ever [31], and with the explosion in the number of nano- and pico- satellites being launched, a similar trend is expected in the field of space exploration as well [29]. All of these systems deal with critical resources like human life, currency, and intricate machinery. This only amplifies the need for employing formal methods to guarantee their correctness.

Verification of distributed algorithms continues to pose a demanding challenge to computer scientists, exacerbated by the fact that paper proofs of these algorithms cannot be trusted [33]. The usual line of reasoning in static analysis

* This work was supported in part by NSF grants CCF-1414078, CCF-1248184, and CNS-1421893, ONR grant N000141512208, and AFOSR grant FA9550-14-1-0261. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of these agencies.

of such systems involves manually writing invariants and then using theorem provers to verify that the invariants follow from the specification and that they imply correctness.

A distributed system comprises a set of processes communicating with each other by message passing while performing local actions that may be triggered upon receiving a set of messages and may conclude with sending a set of messages [14,15]. As such, data processed by any distributed process fall into two categories: (i) *History Variables:* Sets of all messages sent and received[1] and (ii) *Derived Variables:* Local data maintained for efficient computation. Derived variables are often used to maintain results of aggregate queries over sent and received messages.

While reading and writing pseudocode, derived variables are helpful because instead of writing the definition of the variable everywhere, the variable is used instead. Human readers would recall the definition and convince themselves how the algorithm works. While this approach works well for humans, the same is not true for provers. For specifications written with derived variables, invariants have to be added to their proofs which, at the very least, establish that the derived variable implements its definition.

One reason to use derived variables in formal specifications is their existence in pseudocode. Another reason is the lack of high-level languages that provide elegant support for quantifications, history variables, and automatic optimal maintenance of aggregate queries over history variables. The barrier of lack of executable language support for such richness is overcome by high-level languages like DistAlgo [21], which provides native support for history variables, quantifications, and aggregate queries. This motivated us to dispense with derived variables, and study specifications written with only history variables and the impact of this change on their proofs.

Note that uses of history variables provide higher-level specifications of systems in terms of what to compute, as opposed to how to compute with employing and updating derived variables. It makes proofs easier, independent of the logics used for doing the proofs, because important invariants are captured directly in the specifications, rather than hidden under all the incremental updates. On the other hand, it can make model checking much less efficient, just as it can make straightforward execution much less efficient. This is not only because high-level queries are time consuming, but also because maintaining history variables can blow up the state space. This is why automatic incrementalization [27,28,10,23] is essential for efficient implementations, including implementations of distributed algorithms [22,20]. The same transformations for incrementalization can drastically speed up both program execution and model checking.

Contributions. We first describe a systematic style to write specifications of distributed algorithms using message history variables. The only variables in these specifications are the sets of sent and received messages. We show (i) how these are different from the usual pseudocode, (ii) why these are suffi-

[1] This is different than some other references of the term history variables that include sequences of local actions, i.e., execution history [6]

cient for specifying all distributed algorithms, and (iii) when these are better for the provers than other specifications. A method is then explained which, given such specifications, allows us to systematically derive many important invariants which are needed to prove correctness. This method exploits the monotonic increase of the sets of sent and received messages—messages can only be added or read from these sets, not updated or deleted.

We use three existing specifications and their Safety proofs as our case studies: (i) Basic Paxos for single-valued consensus by Lamport et al., distributed as an example with the TLA$^+$ Proof System (TLAPS) [19], (ii) Multi-Paxos for multi-valued consensus [2], and (iii) Multi-Paxos with preemption [2]. Paxos is chosen because it is famous for being a difficult algorithm to grasp, while at the same time it is the core algorithm for distributed consensus—the most fundamental problem in distributed computing. We show that our approach led to significantly reduced sizes of specifications and proofs, numbers of needed manually written invariants, and proof checking times. Our specifications and proofs are available at https://github.com/sachand/HistVar.

Paper Overview. §2 details our style of writing specifications using Basic Paxos as an example. We then describe our strategy to systematically derive invariants in §3 while also showing how using history variables leads to needing fewer invariants. We discuss Multi-Paxos briefly in §4. Results comparing our specifications and proofs with existing work is detailed in §5. §6 concludes with related work.

2 Specifications using message history variables

We demonstrate our approach by developing a specification of Basic Paxos in which we only maintain the set of sent messages. This specification is made to correspond to the specification of Basic Paxos in TLA$^+$ written by Lamport et al. [19]. This is done intentionally to better understand the applicability of our approach. We also simultaneously show Lamport's description of the algorithm in English [17] to aid the comparison, except we rename message types and variable names to match those in his TLA$^+$ specification: *prepare* and *accept* messages are renamed 1a and 2a respectively, their responses are renamed 1b and 2b, respectively, and variable n is renamed b and *bal* in different places.

Distributed consensus. The basic consensus problem, called single-value consensus or single-decree consensus, is to ensure that a single value is chosen from among the values proposed by the processes. The safety requirements for consensus are [17]:

- Only a value that has been proposed may be chosen.
- Only a single value is chosen.

This is formally defined as

$$\mathit{Safety} \triangleq \forall\, v1, v2 \in \mathcal{V} : \phi(v1) \land \phi(v2) \Rightarrow v1 = v2 \tag{1}$$

where $\mathcal{V}$ is the set of possible proposed values, and ϕ is a predicate that given a value v evaluates to true iff v was chosen by the algorithm. The specification of ϕ is part of the algorithm.

Basic Paxos. Paxos solves the problem of consensus. Two main roles of the algorithm are performed by two kinds of processes:

- $\mathcal{P}$ is the set of proposers. These processes propose values that can be chosen.
- $\mathcal{A}$ is the set of acceptors. These processes vote for proposed values. A value is chosen when there are enough votes for it.

A set $\mathcal{Q}$ of subsets of the acceptors, that is $\mathcal{Q} \subseteq 2^{\mathcal{A}}$, is used as a quorum system. It must satisfy the following properties:

- $\mathcal{Q}$ is a set cover for $\mathcal{A}$ — $\bigcup_{Q \in \mathcal{Q}} Q = \mathcal{A}$.
- Any two quorums overlap — $\forall Q1, Q2 \in \mathcal{Q} : Q1 \cap Q2 \neq \emptyset$.

The most commonly used quorum system takes any majority of acceptors as an element in $\mathcal{Q}$. For e.g., if $\mathcal{A} = \{1, 2, 3\}$, then the majority based quorum set is $\mathcal{Q} = \{\{1, 2\}, \{2, 3\}, \{1, 3\}, \{1, 2, 3\}\}$. Quorums are needed because the system can have arbitrary failures. If a process waits for replies from all other processes, as in Two-Phase Commit, the system will hang in the presence of even one failed process. In the mentioned example, the system will continue to work even if acceptor 3 fails because at least one quorum, which is $\{1, 2\}$, is alive.

Basic Paxos solves the problem of single-value consensus. It defines predicate ϕ as

$$\phi(v) \triangleq \exists\, Q \in \mathcal{Q} : \forall\, a \in Q : \exists\, b \in \mathcal{B} : sent(\text{“2b”}, a, b, v) \tag{2}$$

where $\mathcal{B}$ is the set of proposal numbers, also called ballot numbers, which is any set that can be strictly totally ordered. $sent(\text{“2b”}, a, b, v)$ means that a message of type 2b with ballot number b and value v was sent by acceptor a (to some set of processes). An acceptor votes by sending such a message.

Variables. Lamport et al.'s specification of Basic Paxos has four global variables.

- *msgs*— history variable maintaining the set of messages that have been sent. Processes read from or add to this set but cannot remove from it. We rename this to *sent* in both ours and Lamport et al.'s specifications for clarity purposes. This is the only variable maintained in our specifications.
- *maxBal*—for each acceptor, the highest ballot seen by it.
- *maxVBal* and *maxVal*—for each acceptor, *maxVBal* is the highest ballot in which it has voted, and *maxVal* is the value it voted for in that ballot.

Phase 1a. A proposer selects a proposal number b and sends a 1a request with number b to a majority of acceptors.	
Lamport et al.'s	Using *sent* only
$Phase1a(b \in \mathcal{B}) \triangleq$ $\land \nexists\, m \in sent : (m.type = \text{“1a”}) \land (m.bal = b)$ $\land Send([type \mapsto \text{“1a”}, bal \mapsto b])$ $\land$ UNCHANGED $\langle maxVBal, maxBal, maxVal \rangle$	$Phase1a(b \in \mathcal{B}) \triangleq$ $Send([type \mapsto \text{“1a”}, bal \mapsto b])$

Fig. 1. Phase 1a of Basic Paxos

Algorithm steps. The algorithm consists of repeatedly executing two phases. Each phase comprises two actions, one by acceptors and one by proposers.

- **Phase 1a.** Fig. 1 shows Lamport's description in English followed by Lamport et al.'s and our specifications in TLA$^+$. *Send* is an operator that adds its argument to *sent*, i.e., $Send(m) \triangleq sent' = sent \cup \{m\}$.
  1. The first conjunct in Lamport et al.'s specification is not mentioned in the English description and is not needed. Therefore it was removed.
  2. The third conjunct is also removed because the only variable our specification maintains is *sent*, which is updated by *Send*.
- **Phase 1b.** Fig. 2 shows the English description and the specifications of Phase 1b. The first two conjuncts in both specifications capture the precondition in the English description. The remaining conjuncts specify the action.
  1. The first conjunct states that message m received by acceptor a is of type 1a.
  2. The second conjunct ensures that the proposal number *bal* in the 1a message m is higher than that of any 1a request responded to by a. In Lamport et al.'s specification, derived variable $maxBal[a]$ maintains the highest proposal number that a has ever responded to, in 1b and 2b messages, and its second conjunct uses $m.bal > maxBal[a]$. Using *sent* only, we capture this intent more directly, as $\forall\, m2 \in sent : m2.type \in \{\text{“1b”}, \text{“2b”}\} \land m2.acc = a \Rightarrow m.bal > m2.bal$, because those $m2$'s are the response messages that a has ever sent.
  3. The third conjunct is the action of sending a promise (1b message) not to accept any more proposals numbered less than *bal* and with the highest-numbered proposal (if any) that a has accepted, i.e., has sent a 2b message. This proposal is maintained in Lamport et al.'s specification in derived variables *maxVBal* and *maxVal*. We specify this proposal as $max\_prop(a)$, which is either the set of proposals that have the highest proposal number among all accepted by a or if a has not accepted anything, then $\{[bal \mapsto -1, val \mapsto \bot]\}$ as the default, where $-1 \notin \mathcal{B}$ and is smaller than all ballots and $\bot \notin \mathcal{V}$. This corresponds to initialization in Lamport et al.'s specification as shown in Fig. 5.
  4. The remaining conjuncts in Lamport et al.'s specification maintain the variable $maxBal[a]$. A compiler that implements incrementalization [23]

over queries would automatically generate and maintain such a derived variable to optimize the corresponding query.

Phase 1b. If an acceptor receives a 1a request with number *bal* greater than that of any 1a request to which it has already responded, then it responds to the request with a promise not to accept any more proposals numbered less than *bal* and with the highest-numbered proposal (if any) that it has accepted.

Lamport et al.'s	Using *sent* only
$Phase1b(a \in \mathcal{A}) \triangleq$ $\exists\, m \in sent :$ $\wedge m.type = \text{“1a”}$ $\wedge m.bal > maxBal[a]$ $\wedge Send([type \mapsto \text{“1b”},$ $\quad acc \mapsto a, bal \mapsto m.bal,$ $\quad maxVBal \mapsto maxVBal[a],$ $\quad maxVal \mapsto maxVal[a]])$ $\wedge maxBal' =$ $\quad [maxBal \text{ EXCEPT } ![a] = m.bal]$ $\wedge \text{UNCHANGED } \langle maxVBal, maxVal \rangle$	$Phase1b(a \in \mathcal{A}) \triangleq$ $\exists\, m \in sent, r \in max\_prop(a) :$ $\wedge m.type = \text{“1a”}$ $\wedge \forall\, m2 \in sent : m2.type \in \{\text{“1b”}, \text{“2b”}\} \wedge$ $\quad m2.acc = a \Rightarrow m.bal > m2.bal$ $\wedge Send([type \mapsto \text{“1b”},$ $\quad acc \mapsto a, bal \mapsto m.bal,$ $\quad maxVBal \mapsto r.bal,$ $\quad maxVal \mapsto r.val)$ $2bs(a) \triangleq \{m \in sent : m.type = \text{“2b”} \wedge m.acc = a\}$ $max\_prop(a) \triangleq$ $\text{IF } 2bs(a) = \emptyset \text{ THEN } \{[bal \mapsto -1, val \mapsto \bot]\}$ $\text{ELSE } \{m \in 2bs(a) : \forall m2 \in 2bs(a) : m.bal \geq m2.bal\}$

Fig. 2. Phase 1b of Basic Paxos

– **Phase 2a.** Fig. 3 shows Phase 2a. The specifications differ from the English description by using a set of quorums, $\mathcal{Q}$, instead of a majority. The only difference between the two specifications is the removed UNCHANGED conjunct when using *sent* only. It is important to note that the English description fails to mention the first conjunct—a conjunct without which the specification is unsafe. Every 2a message must have a unique ballot.

  Note that the first conjunct in Lamport et. al.'s specification (and therefore ours as well) states that none of the 2a messages sent so far has *bal* equal to b. This is not directly implementable in a real system because this quantification query requires accessing message histories of all processes. We leave this query as is for two main reasons: (i) The focus of this paper is to demonstrate the use of history variables against derived variables and compare them in the light of simpler specification and verification. This removes derived variables but leaves queries on history variables unchanged even though they are not directly implementable. (ii) There is a commonly-used, straightforward, efficient way to implement this query - namely realizing ballot as a tuple in $\mathbb{N} \times \mathcal{P}$ [16]. So a proposer only executes Phase 2a on a ballot proposed by itself (i.e., sent a 1a message with that ballot) and, for efficient implementation, only executes Phase 2a on the highest ballot that it has proposed.

Phase 2a. If the proposer receives a response to its 1a requests (numbered b) from a majority of acceptors, then it sends a 2a request to each of those acceptors for a proposal numbered b with a value v, where v is the value of the highest-numbered proposal among the 1b responses, or is any value if the responses reported no proposals.

Lamport et. al's	Using *sent* only
$Phase2a(b \in \mathcal{B}) \triangleq$ $\land \nexists m \in sent : m.type = \text{“2a”} \land m.bal = b$ $\land \exists\, v \in \mathcal{V}, Q \in \mathcal{Q}, S \subseteq \{m \in sent :$ $\quad m.type = \text{“1b”} \land m.bal = b\} :$ $\quad \land \forall\, a \in Q : \exists\, m \in S : m.acc = a$ $\quad \land \lor \forall\, m \in S : m.maxVBal = -1$ $\quad\quad \lor \exists\, c \in 0..(b-1) :$ $\quad\quad\quad \land \forall\, m \in S : m.maxVBal \leq c$ $\quad\quad\quad \land \exists\, m \in S : \land m.maxVBal = c$ $\quad\quad\quad\quad\quad \land m.maxVal = v$ $\quad \land Send([type \mapsto \text{“2a”}, bal \mapsto b, val \mapsto v])$ $\land \text{UNCHANGED } \langle maxBal, maxVBal,$ $\quad maxVal \rangle$	$Phase2a(b \in \mathcal{B}) \triangleq$ $\land \nexists m \in sent : m.type = \text{“2a”} \land m.bal = b$ $\land \exists\, v \in \mathcal{V}, Q \in \mathcal{Q}, S \subseteq \{m \in sent :$ $\quad m.type = \text{“1b”} \land m.bal = b\} :$ $\quad \land \forall\, a \in Q : \exists\, m \in S : m.acc = a$ $\quad \land \lor \forall\, m \in S : m.maxVBal = -1$ $\quad\quad \lor \exists\, c \in 0..(b-1) :$ $\quad\quad\quad \land \forall\, m \in S : m.maxVBal \leq c$ $\quad\quad\quad \land \exists\, m \in S : \land m.maxVBal = c$ $\quad\quad\quad\quad\quad \land m.maxVal = v$ $\quad \land Send([type \mapsto \text{“2a”}, bal \mapsto b, val \mapsto v])$

Fig. 3. Phase 2a of Basic Paxos

– **Phase 2b.** Fig. 4 shows Phase 2b. Like Phase 1b, we replace the second conjunct with the corresponding query over *sent* and remove updates to the derived variables.

Phase 2b. If an acceptor receives an 2a request for a proposal numbered *bal*, it accepts the proposal unless it has already responded to a 1a request having a number greater than *bal*.

Lamport et al.'s	Using *sent* only
$Phase2b(a \in \mathcal{A}) \triangleq$ $\exists\, m \in sent :$ $\land m.type = \text{“2a”}$ $\land m.bal \geq maxBal[a]$ $\land Send([type \mapsto \text{“2b”}, acc \mapsto a,$ $\quad bal \mapsto m.bal, val \mapsto m.val])$ $\land maxBal' = [maxBal \text{ EXCEPT } ![a] = m.bal]$ $\land maxVBal' = [maxBal \text{ EXCEPT } ![a] = m.bal]$ $\land maxVal' = [maxVal \text{ EXCEPT } ![a] = m.val]$	$Phase2b(a \in \mathcal{A}) \triangleq$ $\exists\, m \in sent :$ $\land m.type = \text{“2a”}$ $\land \forall\, m2 \in sent : m2.type \in \{\text{“1b”}, \text{“2b”}\} \land$ $\quad m2.acc = a \Rightarrow m.bal \geq m2.bal$ $\land Send([type \mapsto \text{“2b”}, acc \mapsto a,$ $\quad bal \mapsto m.bal, val \mapsto m.val])$

Fig. 4. Phase 2b of Basic Paxos

Complete algorithm specification. To complete the algorithm specification, we define, and compare, *vars*, *Init*, *Next*, and *Spec* which are typical TLA$^+$ operator names for the set of variables, the initial state, possible actions leading to the next state, and the system specification, respectively, in Fig. 5.

Lamport et al.'s initialization of $maxVBal$ and $maxVal$ to -1 and $\perp$, respectively, is moved to our definition of max_prop in Fig. 2. We do not need initialization of $maxBal$ because if no 1b or 2b messages have been sent, the universally quantified queries over them would be vacuously true. In Lamport et al.'s specification, this is achieved by initializing $maxBal$ to -1, which is smaller than all ballots, and thus, the conjunct $m.bal > maxBal[a]$ in Fig. 2 holds for the first 1a message received.

Lamport et al.'s	Using *sent* only
$vars \triangleq \langle sent, maxBal, maxVBal, maxVal \rangle$	$vars \triangleq \langle sent \rangle$
$Init \triangleq \land sent = \emptyset$	$Init \triangleq sent = \emptyset$
$\land maxVBal = [a \in \mathcal{A} \mapsto -1]$	
$\land maxBal = [a \in \mathcal{A} \mapsto -1]$	
$\land maxVal = [a \in \mathcal{A} \mapsto \perp]$	
$Next \triangleq \lor \exists\, b \in \mathcal{B} : Phase1a(b) \lor Phase2a(b)$	
$\lor \exists\, a \in \mathcal{A} : Phase1b(a) \lor Phase2b(a)$	
$Spec \triangleq Init \land \Box[Next]_{vars}$	

Fig. 5. Complete Algorithm specification

3 Invariants and proofs using message history variables

Invariants of a distributed algorithm can be categorized into the following three kinds:

1. *Type invariants.* These ensure that all data processed in the algorithm is of valid type. For example, messages of type 1a must have a field $bal \in \mathcal{B}$. If an action sends a 1a message with bal missing or $bal \notin \mathcal{B}$, a type invariant is violated.
2. *Message invariants.* These are invariants defined on message history variables. For example, each message of type 2a has a unique bal. This is expressed by the invariant $\forall\, m1, m2 \in sent : m1.type = \text{“2a”} \land m2.type = \text{“2a”} \land m1.bal = m2.bal \Rightarrow m1 = m2$.
3. *Process invariants.* These state properties about the data maintained in derived variables. For example, in Lamport et al.'s specification, one such invariant is that for any acceptor a, $maxBal[a] \geq maxVBal[a]$.

Fig. 6 shows and compares all invariants used in Lamport et al.'s proof vs. ours. The following operators are used in the invariants for brevity:

$$
\begin{array}{ll}
VotedForIn(a, v, b) \triangleq \exists\, m \in sent : & \\
\quad m.type = \text{“2b”} \wedge m.acc = a \wedge m.val = v \wedge m.bal = b & \\
WontVoteIn(a, b) \triangleq \forall\, v \in \mathcal{V} : \neg VotedForIn(a, v, b) \wedge & \text{– Lamport et al.'s} \\
\quad maxBal[a] > b & \\
WontVoteIn(a, b) \triangleq \forall\, v \in \mathcal{V} : \neg VotedForIn(a, v, b) \wedge & \text{– Using } sent \text{ only} \\
\quad \exists\, m \in sent : m.type \in \{\text{“1b”}, \text{“2b”}\} \wedge m.acc = a \wedge m.bal > b & \\
SafeAt(v, b) \triangleq \forall\, b2 \in 0..(b-1) : \exists\, Q \in \mathcal{Q} : \forall\, a \in Q : & \\
\quad VotedForIn(a, v, b2) \vee WontVoteIn(a, b2) &
\end{array}
\tag{3}
$$

	Lamport et al.'s proof	Our proof
Type Invariants	(I1) $sent \subseteq Messages$ (I2) $maxVBal \in [\mathcal{A} \to \mathcal{B} \cup \{-1\}]$ (I3) $maxBal \in [\mathcal{A} \to \mathcal{B} \cup \{-1\}]$ (I4) $maxVal \in [\mathcal{A} \to \mathcal{V} \cup \{\bot\}]$	$sent \subseteq Messages$
Process Invariants $\forall\, a \in \mathcal{A}$	(I5) $maxBal[a] \geq maxVBal[a]$ (I6) $maxVal[a] = \bot \Leftrightarrow maxVBal[a] = -1$ (I7) $maxVBal[a] \geq 0 \Rightarrow$ $\quad VotedForIn(a, maxVal[a], maxVBal[a])$ (I8) $\forall\, b \in \mathcal{B} : b > maxVBal[a] \Rightarrow$ $\quad \nexists v \in \mathcal{V} : VotedForIn(a, v, b)$	
Message Invariants $\forall m \in sent$	(I9) $m.type = \text{“2b”} \Rightarrow m.bal \leq maxVBal[m.acc]$ (I10) $m.type = \text{“1b”} \Rightarrow m.bal \leq maxBal[m.acc]$	
	(I11) $m.type = \text{“1b”} \Rightarrow$ $\vee \wedge m.maxVal \in \mathcal{V} \wedge m.maxVBal \in \mathcal{B}$ $\quad \wedge VotedForIn(m.acc,$ $\qquad m.maxVal, m.maxVBal)$ $\vee m.maxVBal = -1 \wedge m.maxVal = \bot$	$m.type = \text{“1b”} \Rightarrow$ $\vee VotedForIn(m.acc,$ $\quad m.maxVal, m.maxVBal)$ $\vee m.maxVBal = -1$
	(I12) $m.type = \text{“1b”} \Rightarrow$ $\quad \forall\, b2 \in m.maxVBal + 1..m.bal - 1 : \nexists v \in \mathcal{V} : VotedForIn(m.acc, v, b2)$ (I13) $m.type = \text{“2a”} \Rightarrow SafeAt(m.val, m.bal)$ (I14) $m.type = \text{“2a”} \Rightarrow$ $\quad \forall\, m2 \in sent : m2.type = \text{“2a”} \wedge m2.bal = m.bal \Rightarrow m2 = m$ (I15) $m.type = \text{“2b”} \Rightarrow$ $\quad \exists\, m2 \in sent : m2.type = \text{“2a”} \wedge m2.bal = m.bal \wedge m2.val = m.val$	

Fig. 6. Comparison of invariants. Our proof does not need I2-I10, and needs only I1, a simpler I11, and I12-I15.

Type invariants reduced to one. Lamport et al. define four type invariants, one for each variable they maintain. *Messages* is the set of all possible valid messages. We require only one, (I1). This invariant asserts that the type of all sent messages is valid. (I2-4) are not applicable to our specification.

Process invariants not needed. Lamport et al. define four process invariants, (I5-8), regarding variables $maxVal$, $maxVBal$, and $maxBal$. They are not applicable to our specification, and need not be given in our proof.

(I5) Because $maxBal[a]$ is the highest ballot ever seen by a and $maxVBal[a]$ is the highest ballot a has voted for, we have

$$\begin{aligned} maxBal[a] &= \texttt{max}(\{m.bal : m \in sent \land m.type \in \{\text{“1b”}, \text{“2b”}\} \land m.acc = a\}) \\ maxVBal[a] &= \texttt{max}(\{m.bal : m \in sent \land m.type \in \{\text{“2b”}\} \land m.acc = a\}) \end{aligned} \tag{4}$$

where $\texttt{max}(S) \triangleq$ CHOOSE $e \in S \cup \{-1\} : \forall f \in S : e \geq f$. Note that `max` is not in TLA$^+$ and has to be user-defined. Invariant (I5) is needed in Lamport et al.'s proof but not ours because they use derived variables whereas we specify the properties directly. For example, for Lamport et al.'s Phase 1b, one cannot deduce $m.bal > maxVBal[a]$ without (I5), whereas in our Phase 1b, definitions of $2bs$ and max_prop along with the second conjunct are enough to deduce it.

(I6) Lamport et al.'s proof needs this invariant to prove (I11). Because the initial values are part of *Init* and are not explicitly present in their Phase 1b, this additional invariant is needed to carry this information along. We include the initial values when specifying the action in Phase 1b and therefore do not need such an invariant.

(I7) This invariant is obvious from the definition of *VotedForIn* in Equation (3) and property of $maxVBal$ in Equation (4). The premise $maxVBal[a] \geq 0$ is needed by Lamport et al.'s proof to differentiate from the initial value -1 of $maxVBal[a]$.

(I8) This states that a has not voted for any value at a ballot greater than $maxVBal[a]$. This invariant need not be manually given in our proofs because it is implied from the definition of $maxVBal[a]$.

Message invariants not needed or more easily proved. Before detailing the message invariants, we present a systematic method that can derive several useful invariants used by Lamport et al. and thus make the proofs easier. This method is based on the following properties of our specifications and distributed algorithms:

1. *sent* monotonically increases, i.e., the only operations on it are read and add.
2. Message invariants hold for each sent message of some type, i.e., they are of the form $\forall\, m \in sent : m.type = \tau \Rightarrow \Phi(m)$, or more conveniently if we define $sent_\tau = \{m \in sent : m.type = \tau\}$, we have $\forall\, m \in sent_\tau : \Phi(m)$.
3. $sent = \emptyset$ initially, so the message invariants are vacuously true in the initial state of the system.

4. Distributed algorithms usually implement a logical clock for ordering two arbitrary messages. In Paxos, this is done by ballots.

We demonstrate our method by deriving (I15). The method is applied for each message type used in the algorithm. Invariant (I15) is about 2b messages. We first identify all actions that send 2b messages and then do the following:

1. **Increment.** 2b messages are sent in Phase 2b as specified in Fig. 4. We first determine the increment to $sent$, $\Delta(sent)$, the new messages sent in Phase 2b. We denote a message in $\Delta(sent)$ by δ for brevity. We have, from Fig. 4,

$$\delta = [type \mapsto \text{“2b”}, acc \mapsto a, bal \mapsto m.bal, val \mapsto m.val] \tag{5}$$

2. **Analyze.** We deduce properties about the messages in $\Delta(sent)$. For 2b messages, we deduce the most straightforward property that connects the contents of messages in $\Delta(sent)$ with the message m, from Fig. 4,

$$\phi(\delta) = \exists\, m \in sent : m.type = \text{“2a”} \wedge \delta.bal = m.bal \wedge \delta.val = m.val \tag{6}$$

3. **Integrate.** Because (i) $sent$ monotonically increases, and (ii) ϕ is an existential quantification over $sent$, ϕ holds for all increments to $sent_{2b}$. Property (i) means that once the existential quantification in ϕ holds, it holds forever. Integrating both sides of Equation (6) in the space of 2b messages yields (I15), i.e.,

$$\begin{aligned}\Phi(sent_{2b}) = \forall\, m2 \in sent_{2b} : \exists\, m \in sent : m.type = \text{“2a”} \wedge \\ m2.bal = m.bal \wedge m2.val = m.val\end{aligned} \tag{7}$$

The case for ϕ being universally quantified over $sent$ is discussed with invariant (I12).

Other message invariants. (I9) and (I10) follow directly from Equation (4) and need not be manually specified for our proof. We also derive (I11), (I12), and (I14) as describe in the following.

(I11) Like (I15), (I11) can also be systematically derived, from our Phase 1b in Fig. 2. This invariant is less obvious when variables $maxVal$ and $maxVBal$ are explicitly used and updated because (i) they are not updated in the same action that uses them, requiring additional invariants to carry their meaning to the proofs involving the actions that use them, and (ii) it is not immediately clear if these variables are being updated in Lamport et al.'s Phase 2b in Fig. 4 because a 2b message is being sent or because a 2a message was received.

(I12) To derive (I11) and (I15), we focused on *where* the contents of the new message come from. For (I12), we analyze *why* those contents were chosen. From our Phase 1b with definitions of $2bs$ and max_prop in Fig. 2, we have

$$\begin{aligned}&\phi(\delta) = \\ &\quad \vee \wedge \exists\, m \in sent : m.type = \text{“2b”} \wedge m.acc = \delta.acc \\ &\qquad \wedge \forall\, m \in sent : m.type = \text{“2b”} \wedge m.acc = \delta.acc \Rightarrow \delta.maxVBal \geq m.bal \\ &\quad \vee \wedge \nexists m \in sent : m.type = \text{“2b”} \wedge m.acc = \delta.acc \wedge \delta.maxVBal = -1\end{aligned} \tag{8}$$

ϕ has two disjuncts—the first has a universal quantification and the second has negated existential, which is universal in disguise. If *sent* is universally quantified, integration like for (I15) is not possible because the quantification only holds *at the time of the action*. As new messages are sent in the future, the universal may become violated.

The key is the phrase *at the time*. One way to work around the universal is to add a time field in each message and update it in every action as a message is sent, like using a logical clock. Then, a property like $\phi(\delta) = \forall\, m \in sent_\tau : \psi(m)$ can be integrated to obtain

$$\Phi(sent_\tau) = \forall\, m2 \in sent_\tau : \forall\, m \in sent : m.time < m2.time \Rightarrow \psi(m) \tag{9}$$

Because ballots act as the logical clock in Paxos, we do not need to specify a separate logical clock and we can perform the above integration on Equation (8) to obtain the invariant (I12).

(I14) This invariant is of the form $\forall\, m1, m2 \in sent_\tau, t : \psi(m1, t) \wedge \psi(m2, t) \Rightarrow m1 = m2$. In this case, $\psi(m, t) \triangleq m.bal = t$. Deriving invariants like (I14) is nontrivial unless ψ is already known. In some cases, ψ can be guessed. The intuition is to look for a universal quantification (or negated existential) in the specification of an action. The ideal case is when the quantification is on the message type being sent in the action. Potential candidates for ψ may be hidden in such quantifications. Moreover, if message history variables are used, these quantifications are easier to identify.

Starting with a guess of ψ, we identify the change in the counting measure (cardinality) of the set $\{t : m \in sent_\tau \wedge \psi(m, t)\}$ along with that of $sent_\tau$. In the case of (I14), we look for $\Delta(|\{m.bal : m \in sent_{2a}\}|)$. From our Phase 2a in Fig. 3, we have

$$\begin{aligned} &\Delta(\{m.bal : m \in sent_{2a}\}) = \{b\} \\ &\phi(\Delta(\{m.bal : m \in sent_{2a}\})) = \nexists m \in sent : m.type = \text{“2a”} \wedge m.bal = b \end{aligned} \tag{10}$$

Rewriting ϕ as $\{b\} \not\subseteq \{m.bal : m \in sent_{2a}\}$, it becomes clear that $\Delta(|\{m.bal : m \in sent_{2a}\}|) = 1$. Meanwhile, $\Delta(|\{m \in sent_{2a}\}|) = 1$. Because the counting measure increases by the same amount for both, (I14) can be derived safely.

4 Multi-Paxos

Specification. We have developed new specifications of Multi-Paxos and Multi-Paxos with Preemption that use only message history variables, by removing derived variables from the specifications described in Chand et al. [2]. This is done in a way similar to how we removed derived variables from Lamport et al.'s specification of Basic Paxos.

The most interesting action here was preemption. With preemption, if an acceptor receives a 1a or 2a message with *bal* smaller than the highest that it has seen, it responds with a preempt message that contains the highest ballot

seen by the acceptor. Upon receiving such a message, the receiving proposer would pick a new ballot that is higher than the ballots of all received preempt messages.

This is a good opportunity to introduce the other message history variable, *received*, the set of all messages received. It is different from *sent* because a message could be delayed indefinitely before being received, if at all. In [2], derived variable *proBallot* is introduced to maintain the result of this query on received messages. We contrast this with our new specification in Fig. 7. $Receive(m)$ adds message m to *received*, i.e., $Receive(m) \triangleq received' = received \cup \{m\}$.

Chand et al. [2]	Using *sent* and *received*
$NewBallot(c \in \mathcal{B}) \triangleq$ CHOOSE $b \in \mathcal{B} : b > c \wedge$ $\nexists\, m \in sent : m.type = \text{“1a”} \wedge m.bal = b$ $Preempt(p \in \mathcal{P}) \triangleq \exists\, m \in sent :$ $\wedge\, m.type = \text{“preempt”} \wedge m.to = p$ $\wedge\, m.bal > proBallot[p]$ $\wedge\, proBallot' = [proBallot$ EXCEPT $![p] =$ $\quad NewBallot(m.bal)]$ $\wedge$ UNCHANGED $\langle sent, aVoted, aBal \rangle$ $Phase1a(p \in \mathcal{P}) \triangleq$ $\wedge\, \nexists\, m \in sent : (m.type = \text{“1a”}) \wedge$ $\quad (m.bal = proBallot[p])$ $\wedge\, Send([type \mapsto \text{“1a”},$ $\quad from \mapsto p, bal \mapsto proBallot[p]])$ $\wedge$ UNCHANGED $\langle aVoted, aBal, proBallot \rangle$	$Phase1a(p \in \mathcal{P}) \triangleq \exists\, b \in \mathcal{B} :$ $\wedge \vee \exists\, m \in sent :$ $\quad \wedge\, m.type = \text{“preempt”} \wedge m.to = p$ $\quad \wedge\, Receive(m)$ $\quad \wedge \forall\, m2 \in received' : m2.to = p \wedge$ $\quad\quad m2.type = \text{“preempt”} \Rightarrow b > m2.bal$ $\vee \wedge \nexists\, m \in sent : m.type = \text{“1a”} \wedge$ $\quad\quad m.from = p$ $\quad \wedge$ UNCHANGED $\langle received \rangle$ $\wedge\, Send([type \mapsto \text{“1a”}, from \mapsto p, bal \mapsto b])$

Fig. 7. Preemption in Multi-Paxos

Verification. While we observed a 27% decrease in proof size for Basic Paxos, for Multi-Paxos this decrease was 48%. Apart from the points described in §3, an important player in this decrease was the removal of operator *MaxVotedBallotInSlot* from Chand et al.'s specifications. This operator was defined as

$$MaxVotedBallotInSlot(D, s) \triangleq \mathsf{max}(\{d.bal : d \in \{d \in D : d.slot = s\}\})$$

Five lemmas were needed in Chand et al.'s proof to assert basic properties of the operator. For example, lemma *MVBISType* stated that if $D \subseteq [bal : \mathcal{B}, slot : \mathcal{S}, val : \mathcal{V}]$, then the result of the operator is in $\mathcal{B} \cup \{-1\}$. Removing these lemmas and their proofs alone resulted in a decrease of about 100 lines (about 10%) in proof size.

5 Results

Table 8 summarizes the results of our specifications and proofs that use only message history variables, compared with those by Lamport et al. and Chand et al. We observe an improvement of around 25% across all stats for Basic Paxos and a staggering 50% for Multi-Paxos and Multi-Paxos with Preemption. Following, we list some important results:

- The specification size decreased by 13 lines (25%) for Basic Paxos, from 52 lines for Lamport et al.'s specification to 39 lines for ours. For Multi-Paxos, the decrease is 36 lines (46%), from 78 lines for Chand et al.'s to 42 lines for ours, and for Multi-Paxos with Preemption, the decrease is 45 lines (46%), from 97 to 52.
- The total number of manually written invariants decreased by 54% overall—by 9 (60%) from 15 to 6 for Basic Paxos, by 8 (50%) from 16 to 8 for Multi-Paxos, and by 9 (53%) from 17 to 8 for Multi-Paxos with Preemption. This drastic decrease is because we do not maintain the variables *maxBal*, *maxVBal*, and *maxVal* as explained in §3.
- The proof size for Basic Paxos decreased by 83 lines (27%), from 310 to 227. This decrease is attributed to the fact that our specification does not use other state variables besides *sent*. This decrease is 468 lines (47%), from 988 to 520, for Multi-Paxos, and is 494 lines (48%), from 1032 to 538 for Multi-Paxos with Preemption.
- Proof by contradiction is used twice in the proof by Lamport et al. and thrice for the proofs in Chand et al. We were able to remove all of these because our specification uses queries as opposed to derived variables. The motive behind removing proofs by contradiction is to have easier to understand constructive proofs.
- The total number of proof obligations decreased by 46% overall—by 57 (24%) from 239 to 182 for Basic Paxos, by 450 (49%) from 918 to 468 for Multi-Paxos, and by 468 (49%) from 959 to 491 for Multi-Paxos with Preemption.
- The proof-checking time decreased by 11 seconds (26%), from 42 to 31 for Basic Paxos. For Multi-Paxos and Multi-Paxos with Preemption, TLAPS took over 3 minutes for the proofs in [2] and failed (due to updates in the new version of TLAPS) to check the proofs of about 5 obligations. In contrast, our proofs were able to be checked completely in 1.5 minutes or less.

6 Related work and conclusion

History Variables. History variables have been at the center of much debate since they were introduced in the early 1970s [6,5,7]. Owicki and Gries [25] use them in an effort to prove properties of parallel programs, criticized by Lamport in his writings [13]. Contrary to ours, their history variables were ghost or auxiliary variables introduced for the sole purpose of simpler proofs. Our history variables are *sent* and *received*, whose contents are actually processed in all distributed system implementations.

Metric	Basic Paxos			Multi-Paxos			Multi-Paxos w/ Preemption		
	Lam	Us	Decr	Cha	Us	Decr	Cha	Us	Decr
Spec. size excl. comments	52	39	25%	78	42	46%	97	52	46%
# invariants	15	6	60%	16	8	50%	17	8	53%
# type invariants	4	1	75%	4	1	75%	5	1	80%
# process invariants	4	0	100%	4	0	100%	4	0	100%
# message invariants	7	5	29%	8	7	13%	8	7	13%
Proof size excl. comments	310	227	27%	988	520	47%	1032	538	48%
Type invariants' proof size	22	21	5%	54	34	37%	75	38	49%
Process invariants' proof size	27	0	100%	136	0	100%	141	0	100%
1b† invariants' proof size	21	15	29%	133	70	47%	133	70	47%
2a† invariants' proof size	73	57	22%	264	120	55%	269	120	55%
2b† invariants' proof size	14	12	14%	94	73	22%	94	73	22%
# proofs by contradiction	2	0	100%	3	0	100%	3	0	100%
# obligations in TLAPS	239	182	24%	918	468	49%	959	491	49%
Type inv proof obligations	17	17	0%	69	52	25%	100	60	40%
Process inv proof obligations	39	0	100%	163	0	100%	173	0	100%
1b† inv proof obligations	12	10	17%	160	80	50%	160	80	50%
2a† inv proof obligations	62	52	16%	241	145	40%	249	145	42%
2b† inv proof obligations	9	9	0%	77	44	43%	77	44	43%
TLAPS check time (seconds)	42	31	26%	>191*	80	>58%	>208*	90	>57%

Fig. 8. Summary of results. Lam is from Lamport et al., Cha is from Chand et al. [2], Us is ours in this paper, and Decr is percentage of decrease by ours. Specification size and proof size are measured in lines. An obligation is a condition that TLAPS checks. The time to check is on an Intel i7-4720HQ 2.6 GHz CPU with 16 GB of memory, running 64-bit Windows 10 Home (v1709 b16299.98) and TLAPS 1.5.4. * indicates that the new version of TLAPS failed to check the proof and gave up on checking after that number of seconds. † (I10)–(I12) are 1b invariants, (I13) and (I14) are 2a invariants, and (I9) and (I15) are 2b invariants for Basic Paxos.

Recently, Lamport and Merz [18] present rules to add history variables, among other auxiliary variables, to a low-level specification so that a refinement mapping from a high-level one can be established. The idea is to prove invariants in the high-level specification that serves as an abstraction of the low-level specification. In contrast, we focus on high-level specifications because our target executable language is DistAlgo, and efficient lower-level implementations can be generated systematically from high-level code.

Specification and Verification. Several systems [30,8,4], models [32,3,24], and methods [26,11,12,1] have been developed in the past to specify distributed algorithms and mechanically check proofs of the safety and liveness properties of the algorithms. This work is orthogonal to them in the sense that the idea of maintaining only message history variables can be incorporated in their specifications as well.

Closer to us in terms of the specification is the work by Padon et al. [26], which does not define any variables and instead defines predicate relations which

would correspond to manipulations of our history variables. For example, $Send([type \mapsto$ "1a", $bal \mapsto b])$ is denoted by $start\_round\_msg(b)$. Instead of using TLA$^+$, the temporal logic of actions, they specify Paxos in first-order logic to later exploit benefits of Effectively Propositional Logic, such as satisfiability being decidable in it.

In contrast, we present a method to specify distributed algorithms using history variables, implementable in high-level executable languages like DistAlgo, and then show (i) how such specifications require fewer invariants for proofs and (ii) how several important invariants can be systematically derived.

References

1. Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991.
2. Saksham Chand, Yanhong A Liu, and Scott D Stoller. Formal verification of multi-paxos for distributed consensus. In *FM 2016: Formal Methods: 21st International Symposium, Limassol, Cyprus, November 9-11, 2016, Proceedings 21*, pages 119–136. Springer, 2016.
3. Bernadette Charron-Bost and André Schiper. The Heard-Of model: computing in distributed systems with benign faults. *Distributed Computing*, 22(1):49–71, 2009.
4. Kaustuv Chaudhuri, Damien Doligez, Leslie Lamport, and Stephan Merz. The TLA$^+$ proof system: Building a heterogeneous verification platform. In *Proceedings of the 7th International colloquium conference on Theoretical aspects of computing*, pages 44–44. Springer-Verlag, 2010.
5. Edmund M Clarke. Proving correctness of coroutines without history variables. *Acta Informatica*, 13(2):169–188, 1980.
6. Maurice Clint. Program proving: coroutines. *Acta informatica*, 2(1):50–63, 1973.
7. Maurice Clint. On the use of history variables. *Acta Informatica*, 16(1):15–30, 1981.
8. Cezara Drăgoi, Thomas A Henzinger, and Damien Zufferey. PSync: a partially synchronous language for fault-tolerant distributed algorithms. *ACM SIGPLAN Notices*, 51(1):400–415, 2016.
9. Mario Gerla, Eun-Kyu Lee, Giovanni Pau, and Uichin Lee. Internet of vehicles: From intelligent grid to autonomous cars and vehicular clouds. In *Internet of Things (WF-IoT), 2014 IEEE World Forum on*, pages 241–246. IEEE, 2014.
10. Michael Gorbovitski. *A system for invariant-driven transformations*. PhD thesis, Computer Science Department, Stony Brook University, 2011.
11. Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R Lorch, Bryan Parno, Michael L Roberts, Srinath Setty, and Brian Zill. IronFleet: proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 1–17. ACM, 2015.
12. Philipp Küfner, Uwe Nestmann, and Christina Rickmann. Formal verification of distributed algorithms. *Theoretical Computer Science*, pages 209–224, 2012.
13. Leslie Lamport. My writings :: Proving the correctness of multiprocess programs. https://lamport.azurewebsites.net/pubs/pubs.html. Accessed: 2017-10-10.
14. Leslie Lamport. The implementation of reliable distributed multiprocess systems. *Computer Networks (1976)*, 2(2):95–114, 1978.
15. Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(3):872–923, 1994.

16. Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.
17. Leslie Lamport. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.
18. Leslie Lamport and Stephan Merz. Auxiliary variables in TLA$^+$. *ArXiv e-prints*, March 2017.
19. Leslie Lamport, Stephan Merz, and Damien Doligez. Paxos.tla. https://github.com/tlaplus/v1-tlapm/blob/master/examples/paxos/Paxos.tla. Last modified Fri Nov 28 10:39:17 PST 2014 by Lamport. Accessed Feb 6, 2018.
20. Yanhong A Liu, , Jon Brandvein, Scott D Stoller, and Bo Lin. Demand-driven incremental object queries. In *Proceedings of the 18th International Symposium on Principles and Practice of Declarative Programming (PPDP)*, pages 228–241. ACM, 2016.
21. Yanhong A Liu, Scott D Stoller, and Bo Lin. From clarity to efficiency for distributed algorithms. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 39(3):12, 2017.
22. Yanhong A Liu, Scott D Stoller, Bo Lin, and Michael Gorbovitski. From clarity to efficiency for distributed algorithms. In *Proceedings of the 27th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 395–410. ACM, 2012.
23. Yanhong Annie Liu. *Systematic Program Design: From Clarity To Efficiency*. Cambridge University Press, 2013.
24. Nancy A Lynch and Mark R Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, pages 137–151. ACM, 1987.
25. Susan Owicki and David Gries. An axiomatic proof technique for parallel programs i. *Acta informatica*, 6(4):319–340, 1976.
26. Oded Padon, Giuliano Losa, Mooly Sagiv, and Sharon Shoham. Paxos made EPR: decidable reasoning about distributed protocols. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):108, 2017.
27. Robert Paige and Shaye Koenig. Finite differencing of computable expressions. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):402–454, 1982.
28. Tom Rothamel and Yanhong A Liu. Generating incremental implementations of object-set queries. In *Proceedings of the 7th international conference on Generative programming and component engineering*, pages 55–66. ACM, 2008.
29. Klaus Schilling. Perspectives for miniaturized, distributed, networked cooperating systems for space exploration. *Robotics and Autonomous Systems*, 90:118–124, 2017.
30. Ilya Sergey, James R Wilcox, and Zachary Tatlock. Programming and proving with distributed protocols. *Proceedings of the ACM on Programming Languages*, 2(POPL):28, 2017.
31. Florian Tschorsch and Björn Scheuermann. Bitcoin and beyond: A technical survey on decentralized digital currencies. *IEEE Communications Surveys & Tutorials*, 18(3):2084–2123, 2016.
32. James R Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D Ernst, and Thomas Anderson. Verdi: a framework for implementing and formally verifying distributed systems. In *ACM SIGPLAN Notices*, volume 50, pages 357–368. ACM, 2015.
33. Pamela Zave. Using lightweight modeling to understand Chord. *ACM SIGCOMM Computer Communication Review*, 42(2):49–57, 2012.