Automating Cutoff-based Verification of Distributed Protocols

Shreesha G. Bhat and Kartik Nagar
Department of CSE, IIT Madras
Chennai, India
shreeshagbhat@gmail.com, nagark@cse.iitm.ac.in

***Abstract*—Distributed protocols are generally parametric and are expected to work correctly on systems containing any number of nodes. Therefore, proving their correctness becomes an infinite state verification problem. The usual approach for verifying distributed protocols is to provide an inductive invariant that is strong enough to imply the safety property. But inductive invariants for even simple distributed protocols can be intricate and synthesizing them in an automated manner is a hard problem. In this work, we investigate an orthogonal cutoff-based technique for verifying distributed protocols. In a cutoff-based approach, one provides a finite-sized instance of the system which encompasses all possible modes of violation of the safety property. Analyzing such a cutoff instance for safety violations suffices to prove the correctness of the protocol for all instances. In this work, we formalize a simulation-based approach to check whether a given instance is a cutoff instance for protocols written in a general modelling language (RML) by identifying sufficient conditions which can be efficiently encoded in SMT. We propose simple static analyses to automatically synthesize the cutoff instance, simulation relation and other proof components, thus leading to a fully automated verification procedure. Finally we apply our technique on a number of protocols ranging from simple leader election and mutual exclusion protocols to complex quorum-based consensus protocols.**

I. Introduction

Distributed protocols allow disparate nodes to work together towards completing a task, and form the backbone of today's distributed systems. These protocols are typically specified in a parametric fashion, which means they can be instantiated on a system with any number of nodes. The nodes communicate with each other through message passing, and these messages can be arbitrarily delayed or even lost. However, the distributed protocol is expected to work correctly under all such conditions. Here, correctness is typically defined in terms of a safety property which must be obeyed by every node at every step of the protocol. For example, the safety property of a distributed mutual exclusion protocol would say that two nodes should not be in their critical section at the same time. Since the protocols need to consider every possible network behavior, they are quite complex in nature. Verifying the correctness of distributed protocols then becomes highly important, but this problem is significantly complicated by the parametric nature of the protocol and the asynchronous, non-deterministic nature of the underlying network. Essentially, every possible instantiation of the protocol needs to be proven correct, and each such instantiation itself needs to consider a large number of network behaviors. Further, there could be an infinite number of instantiations of the protocol.

Recent approaches ([1]–[6]) to verifying distributed protocols typically aim to find an inductive invariant, which is a property of the protocol state satisfied at every step of any protocol instance, which is inductive in nature and is stronger than the safety property. However, finding an inductive invariant is very hard, as conceptually, it should encompass all the complex logic that the protocol employs to maintain the safety property *under any abnormal network behavior in any instantiation*. In this work, we consider an alternative cutoff-based approach to protocol verification that cleanly separates the two problems of dealing with *arbitrary instantiations* and *arbitrary network behavior*. This approach requires a cutoff instance with a fixed, finite number of nodes whose correctness implies the correctness of any arbitrary protocol instance. Then, we only need to consider how the protocol maintains the safety property under arbitrary network behavior in the cutoff instance. Further, since the cutoff instance will have a constant, finite number of nodes, verifying its correctness becomes a finite state verification problem, which can be solved in an automated fashion.

In this paper, we focus on the problem of finding such a cutoff instance, and automatically showing that it is indeed a cutoff. The definition of a cutoff instance gives us the following characterization: *if there exists a violation of the safety property in any arbitrary protocol instance, then there should also exist a violation in the cutoff instance*. We automatically construct a cutoff instance which can simulate any violation of the safety property in any arbitrary protocol instance. While this seems like a tall order, we hypothesize that this problem is simpler due to two reasons: (i) a violation of the safety property typically involves only a small number of nodes (for example, a violation of the mutual exclusion property would only require *two* nodes to be in their critical section together), and further, the participation of other nodes of the system is either not required, or can be simulated by the violating nodes themselves, and (ii) most of the complex logic in the protocol implementation which ensures the absence of a violation can be side-stepped, since we are actually interested in simulating the presence of a violation.

While previous works have also attempted to use cutoff based approaches for verification ([7]–[10]), they have mostly been limited to either a restricted class of protocols [8] with strong assumptions on the underlying network or a restricted class of specifications [9]. In this work, we consider a variety of protocols targeting different goals (consensus, mutual exclusion, key-value store, etc.) and do not make any assumptions about the underlying network. Our approach takes as input the protocol description written in the Relational Modeling Language (RML). We first develop a formalization of the cutoff approach which defines sufficient conditions for proving that a given protocol instance is a cutoff instance, which can be encoded using SMT. We then use our hypothesis concerning the simplicity of the cutoff instance to develop a static analysis based approach which directly synthesizes the cutoff instance from a violation of the safety property. Beginning from a state which violates the safety property, our analysis moves backwards to identify the necessary protocol actions and state components that could be involved in a violation. We then use the output of the static analysis to create a cutoff instance which faithfully simulates all the protocol actions and state components which could be involved in a violation. Finally, we apply our SMT encoding to check the correctness of the synthesized cutoff instance. We have implemented the proposed approach and applied it on 8 different distributed protocols, providing a fully automated cutoff-based proof of correctness for all of them.

To summarize, we make the following contributions:

1) We formalize the cutoff approach for distributed protocols written in the RML language, and identify sufficient conditions for proving the correctness of a cutoff instance.
2) We propose a simple static analysis-based approach to automatically synthesize from the protocol description, a cutoff instance and a simulation relation for proving the correctness of the cutoff.
3) We have implemented the approach in a prototype tool and have successfully verified 8 challenging protocols.

The rest of the paper is organized as follows: In §2, we illustrate the cutoff-based approach to protocol verification and our synthesis algorithm using an example. We formalize the cutoff approach for protocols written in RML in §3 and §4. Details of our synthesis algorithm are presented in §5. Experimental results are given in §6, followed by a discussion on related works and conclusion in §7.

II. Motivating Example: The Sharded Key-Value Store

A. Protocol Description

As a motivating example to demonstrate our technique, we consider the sharded key-value store protocol described in [1]. The protocol maintains key-value pairs distributed across a set of nodes. It implements a mechanism for nodes to *reshard* key-value pairs amongst one another in the presence of an unreliable network while maintaining the safety property that no two nodes should ever own a key simultaneously. A detailed pseudocode description of the protocol in the RML language [11] is provided below in Fig. 1.

Algorithm 1 The Sharded Key Value Store Protocol

```
1: type $key, value, node, seqnum$
2: relation $table : node, key, value$
3: relation $transfer\_msg : node, node, key, value, seqnum$
4: relation $ack\_msg : node, node, seqnum$
5: relation $seqnum\_sent : node, seqnum$
6: relation $unacked : node, node, key, value, seqnum$
7: relation $seqnum\_recvd : node, node, seqnum$
8: init $\forall n_1, n_2, k, v_1.\ table(n_1, k, v_1) \wedge table(n_2, k, v_2) \implies n_1 = n_2 \wedge v_1 = v_2$ ▷ All other relations are empty
9: action Reshard($n\_old : node, n\_new : node, k : key, v : value, s : seqnum$)
10:     require $table(n\_old, k, v) \wedge \neg seqnum\_sent(s)$
11:     $seqnum\_sent(s) \leftarrow true$
12:     $table(n\_old, k, v) \leftarrow false$
13:     $transfer\_msg(n\_old, n\_new, k, v, s) \leftarrow true$
14:     $unacked(n\_old, n\_new, k, v, s) \leftarrow true$
15: action DropTransferMsg($src : node, dst : node, k : key, v : value, s : seqnum$)
16:     require $transfer\_msg(src, dst, k, v, s)$
17:     $transfer\_msg(src, dst, k, v, s) \leftarrow false$
18: action Retransmit($src : node, dst : node, k : key, v : value, s : seqnum$)
19:     require $unacked(src, dst, k, v, s)$
20:     $transfer\_msg(src, dst, k, v, s) \leftarrow true$
21: action RecvTransferMsg($src : node, dst : node, k : key, v : value, s : seqnum$)
22:     require $transfer\_msg(src, dst, k, v, s) \wedge \neg seqnum\_recvd(s)$
23:     $seqnum\_recvd(s) \leftarrow true$
24:     $table(dst, k, v) \leftarrow true$
25: action SendAck($src : node, dst : node, k : key, v : value, s : seqnum$)
26:     require $transfer\_msg(src, dst, k, v, s) \wedge seqnum\_recvd(s)$
27:     $ack\_msg(s) \leftarrow true$
28: action DropAckMsg($src : node, dst : node, k : key, v : value, s : seqnum$)
29:     require $ack\_msg(s)$
30:     $ack\_msg(s) \leftarrow false$
31: action RecvAckMsg($src : node, dst : node, k : key, v : value, s : seqnum$)
32:     require $ack\_msg(s)$
33:     $unacked(src, dst, k, v, s) \leftarrow false$
34: action Put($n : node, k : key, v : value$)
35:     require $\exists v'.\ table(n, k, v')$
36:     $table(n, k, *) \leftarrow false$
37:     $table(n, k, v) \leftarrow true$
38: safety $\forall k, n_1, n_2, v_1, v_2, k.\ table(n_1, k, v_1) \wedge table(n_2, k, v_2) \implies n_1 = n_2 \wedge v_1 = v_2$
```

The protocol is described using a set of types, relations and actions. A type (or sort in RML terminology) is defined for nodes, keys, values and sequence numbers. The relations describe the state of the protocol and are defined over these sorts. In a step of the execution, any action can be fired provided that its guard (specified by the **require** keyword) is satisfied.

The relation $table(n, k, v)$ indicates that the node n holds the key k with the value v. A Reshard action generates a $transfer\_msg$ from the key's current owner to its new owner. Transfer messages can be arbitrarily dropped (through the DropTransferMsg action), and hence the protocol employs an acknowledgment mechanism, whereby the new owner needs to send an acknowledgment message upon receiving a *transfer_msg*, and the current owner will keep re-transmitting (through the Retransmit action) until it receives an acknowledgment. The acknowledgement message itself can be dropped and might require re-transmission. Since each *transfer_msg* message is tagged with a unique sequence number, the receiving node can ignore duplicate *transfer_msg*'s that arise from the re-transmission mechanism by marking the sequence number as received in line 24; the absence of which is used as a guard by RecvTransferMsg action. This prevents safety violations that can occur due to older transfer messages entering their out-of-date key value pair into the table of the destination node after it has already been re-sharded to some other node, or subsequent Put actions have occurred thereby altering the associated value.

B. Cutoff based Verification

The safety property for this protocol says that in all runs, we cannot have two different table entries for the same key. Intuitively, this is maintained at all times, because either a single node contains the key in its table, or the key is in-transit. The unique sequence number associated with a *transfer_msg* ensures that re-transmissions do not break the safety property. Prior works [1], [11] construct a complex inductive invariant which leverages the above observation to show the uniqueness of a number of state components, and ultimately implies the safety property. In this work, we take an orthogonal approach where we assume the existence of a hypothetical violation and focus on (1) identifying the key state components and actions of the protocol that contribute to this violation, and (2) simulating this violation by maintaining these state components in a fixed, small protocol instance. If the cutoff instance can be shown to simulate any violation of the safety property, proving the safety of the cutoff instance is sufficient to establish correctness for all instances of the protocol. This essentially formalizes the 'small model' property that has been empirically established by many prior works for bugs in concurrent and distributed systems. Note that while synthesizing the cutoff instance, we can completely ignore how the protocol blocks out potential scenarios where a violation can occur, which is one of the classical hurdles in crafting inductive invariants. For the sharded key-value store protocol, we show that a cutoff instance with 2 nodes can simulate all possible violations in arbitrary sized instances of the protocol (note that size refers to number of nodes).

C. Static Analysis

We employ a static analysis based approach on the protocol description to find out the relevant state components and actions that are necessary for simulating violations of the safety property. Consider a violation in an arbitrary size system L where we have two distinct nodes A_L, B_L and key K such that $table(A_L, K, V_1)$ and $table(B_L, K, V_2)$ hold. We are interested in collecting the relevant state components and actions that are responsible for this violating state of L. At a very high level, our static analysis starts from the state components directly involved in the violation, and then finds actions which can set these state components. However, for these actions to be enabled, their guards will also need to be maintained. So the state components in the guards also now become relevant, and the above process continues until no new relevant actions or state components are found.

For the sharded key value store protocol, we start with the state components that are involved in the violation of the safety property as the initial set of relevant state components, $S = \{table(A_L, K, V_1), table(B_L, K, V_2)\}$. Consider the actions that set the clauses $table(A_L\langle B_L\rangle, K, V_1\langle V_2\rangle)$ (we use entries in brackets $\langle\rangle$ to succinctly represent both the clauses). We find that any action of the type $\mathsf{Put}(A_L\langle B_L\rangle), K, V_1\langle V_2\rangle)$ and $\mathsf{RecvTransferMsg}(*, A_L\langle B_L\rangle, K, V_1\langle V_2\rangle), *)$ can set these $table$ entries, where $*$ represents any value. These are added to the set of relevant actions (denoted by A). Now we consider the components in the guards of these actions. For the RecvTransferMsg actions, the guard contains the clauses $\neg seqnum\_recvd(*)$ and $transfer\_msg(*, A_L\langle B_L\rangle, K, V_1\langle V_2\rangle, *)$. For the Put actions, we have $\exists v.\ table(A_L\langle B_L\rangle, K, v)$ as the guard clause. For the existential quantifier, we include $table(A_L\langle B_L\rangle, K, *)$ where the value entry is not restricted and therefore all such table entries are tracked as relevant. These entries are added to the set S.

In this way, we keep on collecting relevant actions and clauses, terminating in a fixed point after a few iterations. We also simplify the sets by noting that $*$ entries subsume other entries that contain specific values in that field. For example, if the S set contains an entry $table(A_L, K, V_1)$ and also an entry $table(A_L, K, *)$, the latter subsumes the former. On performing such reductions, we get the following fixed point sets S and A

$$\begin{aligned} S = \{&table(*, K, *), transfer\_msg(*, *, K, *, *),\\ &\neg seqnum\_recvd(*), \neg seqnum\_sent(*),\\ &unacked(*, *, K, *, *)\}\\ A = \{&\mathsf{Put}(*, K, *), \mathsf{RecvTransferMsg}(*, *, K, *, *),\\ &\mathsf{Reshard}(*, *, K, *, *), \mathsf{Retransmit}(*, *, K, *, *)\} \end{aligned}$$

Notice that though the protocol has 8 actions in total, the action set obtained from static analysis shows that only 4 of these actions are actually relevant in a violation. In particular, actions such as DropTransferMsg and SendAck are not required to simulate a violation. Intuitively, this is because these actions are not necessary to actually transfer a key from one node to another, which is needed for realizing a potential violation. Secondly, although the correctness of the protocol (that is, avoiding a violation) depends on a complex invariant involving uniqueness of a number of state components, we do not require any of that complexity to simulate a violation. The static analysis essentially ignores how exactly a violating state might have been obtained, but instead tries to trace the state components and actions that are essential for recreating the violation. For example, it is possible that a transfer message may have been dropped by the network in a violating execution, and hence would need to be re-transmitted. However, the cutoff system need not drop the message in the first place (re-transmission is still required). Intuitively, if a violation occurs in L, by maintaining the state components in S and performing only the relevant actions in A, we can recreate the violation in the cutoff system C.

D. Simulation Relation & Lockstep

While the static analysis gives us the relevant state components and actions that need to be maintained in a cutoff system, we still need to prove that any violation in any protocol instance can be simulated by the cutoff instance. To show this, we establish a simulation between any arbitrary instance L and a cutoff instance C. The simulation is primarily governed by a *lockstep* which describes the action(s) taken by the cutoff instance C for every action in L. An action in L is simulated as zero or more actions in C. We also establish a *simulation relation* that holds inductively on the states of both L and C as they progress according to the lockstep. The simulation relation will be strong enough to show that at any step, a violation of the safety property in L will imply a violation in the state of C as well.

The main ingredients of the simulation relation and lockstep have already been identified via the static analysis, i.e. the relevant state components and corresponding actions required to reach a violating state. What remains is to map the relevant state components and actions of L to corresponding components of C. Such a mapping can be obtained by mapping nodes of L to their corresponding simulating node in C. Denoting the node mapping as $sim : \mathcal{D}_L \rightarrow \mathcal{D}_C$ (where $\mathcal{D}_x$ represents the set of nodes in the instance x), the simulation relation maintains that relevant state components from the set S obtained from static analysis corresponding to any node $n \in \mathcal{D}_L$ in L match the corresponding state component of $sim(n)$ in C. The simulation relation does not say anything about the state components which are not relevant for the violation. Similarly, the lockstep ensures that whenever any action from A occurs in L, the corresponding action is triggered in C. The rest of the actions of L are ignored as they are not relevant to simulate the violation.

Specifically, for the sharded key value store protocol, let us denote the two nodes in the cutoff instance as A_C and B_C. Recall that A_L and B_L were nodes of the larger instance L which were involved in the violation. We have $sim(A_L) = A_C$ and $sim(B_L) = B_C$. We map the rest of the nodes to one of A_C or B_C, say B_C i.e. $\forall N \in \mathcal{D}_L . (N \neq A) \wedge (N \neq B) \implies sim(N) = B_C$. Intuitively, a node $N_C \in \mathcal{D}_C$ maintains the state and performs the actions for all the nodes $N_L \in \mathcal{D}_L$ such that $sim(N_L) = N_C$.

Applying the sim mapping on the relevant state components S we get the following 5 clauses in the simulation relation:

(1) $table_L(n, K, v) \implies table_C(sim(n), K, v)$
(2) $unacked_L(n_1, n_2, K, v, s) \implies$
$unacked_C(sim(n_1), sim(n_2), K, v, s)$
(3) $\neg seqnum\_sent_L(s) \implies \neg seqnum\_sent_C(s)$
(4) $\neg seqnum\_recvd_L(s) \implies \neg seqnum\_recvd_C(s)$
(5) $transfer\_msg_L(n_1, n_2, K, v, s) \implies$
$transfer\_msg_C(sim(n_1), sim(n_2), K, v, s)$

Here, we use rel_L and rel_C to denote the relation rel of the protocol for the instances L and C respectively and assume universal quantifiers over all lower-cased variables for each clause. Notice that the simulation relation ensures that any violation of safety property in the protocol state of the larger system (say $table_L(A_L, K, V_1)$ and $table_L(B_L, K, V_2)$) will result in a violation of the cutoff system. The lockstep defines the actions fired in the cutoff instance for actions of the larger instance, and ensures that the above simulation relation is maintained for every step of every execution. For actions not in the lockstep, no action is fired in the cutoff instance. Again, the sim mapping and the relevant actions A give the following lockstep:

(1) $\mathsf{Put}_\mathsf{L}(n, K, c)$ **is simulated as** $\mathsf{Put}_\mathsf{C}(sim(N), K, V)$
(2) $\mathsf{Reshard}_\mathsf{L}(n_1, n_2, K, v, s)$ **is simulated as**
$\mathsf{Reshard}_\mathsf{C}(sim(n_1), sim(n_2), K, v, s)$
(3) $\mathsf{Retransmit}_\mathsf{L}(n_1, n_2, K, v, s)$ **is simulated as**
$\mathsf{Retransmit}_\mathsf{C}(sim(n_1), sim(n_2), K, v, s)$
(4) $\mathsf{RecvTransferMsg}_\mathsf{L}(n_1, n_2, K, v, s)$ **is simulated as**
$\mathsf{RecvTransferMsg}_\mathsf{C}(sim(n_1), sim(n_2), K, v, s)$

Now, we can show that the simulation relation holds inductively as the two instances L and C execute as-per the lockstep. This ensures that for every violating execution of the larger instance L, there exists a violating execution of C. By independently showing that C does not exhibit any violations (which is a much simpler problem, since it has only 2 nodes), we can infer the correctness of the protocol.

III. Setup

We consider distributed protocols written in the Relational Modeling Language (RML) [11]. RML is a Turing-complete language, and has been used in many prior works related to distributed protocol verification. RML uses the notions of *relations* and *functions* as used in many-sorted first order logic to describe the state of a distributed protocol. Further, these can be defined over arbitrary domains, as specified by the protocol developer. Constraints on the initial state of the protocol, as well as the safety property can then be directly encoded as FOL formulae over the declared relations and functions.

The protocol description in RML $\mathtt{P} = \langle \mathtt{D}, \mathtt{R}, \mathtt{F}, \Psi, \mathtt{A}, \Phi \rangle$ consists of a set of declarations ($\mathtt{D}$,$\mathtt{R}$,$\mathtt{F}$), axioms (Ψ), actions ($\mathtt{A}$) and a safety property (Φ). The declarations define the vocabulary: $\mathtt{D}$, $\mathtt{R}$ and $\mathtt{F}$ denote the set of domain names, relation names and function names respectively (along with the relation and function signatures). The axioms (Ψ) are FOL formulae defined over the vocabulary which encode properties of the domains. Φ denotes the safety property, which is another FOL formula, while $\mathtt{A}$ denotes the actions of the protocol.

Given the protocol description, we construct a labeled transition system modeling the execution of the protocol. The transition system $\mathcal{A}^{\mathtt{P}}_{\mathcal{I}} = (\Sigma, \Sigma_0, \delta)$ is parameterized by a domain interpretation function $\mathcal{I}$ which associates a finite domain of values with each domain name $\mathtt{d} \in \mathtt{D}$. For the interpretation function $\mathcal{I}$ to be valid, we require the domains in range of $\mathcal{I}$ to satisfy all the axioms in Ψ. Each state $\sigma \in \Sigma$ is an interpretation of function and relation names in $\mathtt{F}$ and $\mathtt{R}$ to actual functions and relations over the domains defined by the interpretation function $\mathcal{I}$. That is, for a function signature $\mathtt{f} : (\mathtt{d}_1 \times \ldots \mathtt{d}_n) \to \mathtt{d}$ in the protocol description, $\sigma(\mathtt{f})$ will be a function of the form $\mathcal{I}(\mathtt{d}_1) \times \ldots \mathcal{I}(\mathtt{d}_n) \to \mathcal{I}(\mathtt{d})$. The same holds for a relation $\mathtt{r}$ in the description.

The RML protocol description also consists of a set of axioms Ψ_0 constraining the functions and relations in the initial state of the system. We define $\Sigma_0 = \{\sigma \in \Sigma \mid \sigma \models \Psi_0\}$ to be the set of states obeying the initialization axioms. Note that the notation $\sigma \models \Psi$ denotes the standard FOL definition of an interpretation (σ) being the model of an FOL formula (Ψ).

Transitions of $\mathcal{A}^{\mathtt{P}}_{\mathcal{I}}$ will correspond to actions of the protocol. An action $\mathtt{a}(\bar{\mathtt{v}} : \bar{\mathtt{d}}) = \langle g(\bar{\mathtt{v}}), u(\bar{\mathtt{v}})\rangle$ is parameterized over a set of (typed) variable names ($\bar{\mathtt{v}}$), and consists of two components: (i) an FOL formula g (also called the guard) which can contain free variables from $\bar{\mathtt{v}}$, (ii) an FOL formula u which models the change in the protocol state, defined over unprimed and primed versions of the functions and relations of the protocol. If the current state of the protocol obeys the guard, then the state is updated atomically using the update formula. The transitions $\mathcal{A}^{\mathtt{P}}_{\mathcal{I}}$ caused by the action $\mathtt{a}$ in the protocol are formally defined as follows:

$$\delta_{\mathtt{a}} = \{(\sigma, \mathtt{a}(\bar{x}), \sigma') \mid \exists \bar{x} \in \mathcal{I}(\bar{\mathtt{d}}).\ \sigma \models g[\bar{x}/\bar{v}] \wedge \sigma, \sigma' \models u[\bar{x}/\bar{v}]\}\}$$

That is, for every valuation $\bar{x}$ of the variables $\bar{\mathtt{v}}$, there are transitions from states σ which obey the guard g to states σ' such that σ, σ' satisfy the update formula. The transition is labeled by the action name along with the actual parameters, i.e. $\mathtt{a}(\bar{x})$. The complete set of transitions is obtained by considering the transition set of every action of the protocol: $\delta = \cup_{\mathtt{a} \in \mathtt{A}} \delta_{\mathtt{a}}$. Let δ^* denote the reflexive and transitive closure of δ.

The safety property Φ is defined as a FOL formulae using the declared domains, functions and relations. In this work, we assume that Φ only uses universal quantifiers. Hence, Φ has the form : $\forall(\bar{x} : \bar{\mathtt{d}}).\ \phi$. This assumption is consistent with prior works related to distributed protocol verification, and is not restrictive as almost all safety properties can be naturally expressed using just universal quantification.

A trace of $\mathcal{A}^{\mathtt{P}}_{\mathcal{I}}$ is a sequence of states and transition labels of the form $\sigma_0 a_1 \sigma_1 a_2 \sigma_2 \ldots a_n \sigma_n$ such that $\sigma_0 \in \Sigma_0$ and $(\sigma_i, a_{i+1}, \sigma_{i+1}) \in \delta$ for all $i, 0 \leq i \leq n-1$. Let $\mathcal{T}(\mathcal{A}^{\mathtt{P}}_{\mathcal{I}})$ denote the set of traces of $\mathcal{A}^{\mathtt{P}}_{\mathcal{I}}$. We use $[\![\mathcal{A}^{\mathtt{P}}_{\mathcal{I}}]\!]$ to denote the set of reachable states of $\mathcal{A}^{\mathtt{P}}_{\mathcal{I}}$, i.e. $[\![\mathcal{A}^{\mathtt{P}}_{\mathcal{I}}]\!] = \{\sigma' \mid \sigma_0 \ldots \sigma' \in \mathcal{T}(\mathcal{A}^{\mathtt{P}}_{\mathcal{L}})\}$. A transition system is safe if all of reachable states obey the safety property of the protocol:

Definition 1. Given a distributed protocol $\mathtt{P} = \langle \mathtt{D}, \mathtt{R}, \mathtt{F}, \Psi, \Phi, \mathtt{A}\rangle$, a valid interpretation of domains $\mathcal{I}$ obeying Ψ, the transition system $\mathcal{A}^{\mathtt{P}}_{\mathcal{I}}$ is **safe** if for every reachable state $\sigma \in [\![\mathcal{A}^{\mathtt{P}}_{\mathcal{I}}]\!]$, $\sigma \models \Phi$.

While $\mathcal{A}^{\mathtt{P}}_{\mathcal{I}}$ will be a finite state system (because every domain defined by $\mathcal{I}$ is finite), there can in general be infinite number of domains which satisfy the axioms Ψ of the protocol. For a distributed protocol to be safe, the transition system corresponding to every valid domain interpretation should be safe:

Definition 2. A distributed protocol $\mathtt{P} = \langle \mathtt{D}, \mathtt{R}, \mathtt{F}, \Psi, \Phi, \mathtt{A}\rangle$ is safe if for every valid domain interpretation function $\mathcal{I}$ satisfying the axioms Ψ, $\mathcal{A}^{\mathtt{P}}_{\mathcal{I}}$ is safe.

IV. CUTOFF BASED VERIFICATION

Each valid interpretation of the domains of a protocol can be seen as a protocol instance. A typical example of a domain with infinite number of valid interpretations is the domain of nodes participating in a protocol. To prove that a protocol is correct, we would need to show its correctness for all possible protocol instances. In cutoff based verification, the idea is to only show correctness for a specific protocol instance called a cutoff instance. In the following, we now formalize cutoff based verification in our framework.

Definition 3. Given a distributed protocol $\mathtt{P}$, a **cutoff instance** $\mathcal{C}$ is a valid interpretation of domains such that if $\mathcal{A}^{\mathtt{P}}_{\mathcal{C}}$ is safe, then for any valid interpretation $\mathcal{L}$, $\mathcal{A}^{\mathtt{P}}_{\mathcal{L}}$ is safe.

Theorem 1. For a distributed protocol $\mathtt{P}$, if $\mathcal{C}$ is a cutoff instance, and $\mathcal{A}^{\mathtt{P}}_{\mathcal{C}}$ is safe, then the distributed protocol $\mathtt{P}$ is safe.

Proof. The proof follows directly from the definitions of a cutoff instance and safety of a distributed protocol. ☐

Notice that the definition of a cutoff instance implies that if there exists a protocol instance with a violation of the safety property, then the cutoff instance will also have a violation of the safety property. In essence, the cutoff instance can simulate the violation of the safety property in any protocol instance. We use this characterization to propose three conditions which together imply that a protocol instance is a cutoff instance.

These conditions require a simulation relation between states of any arbitrary protocol instance and states of the cutoff instance. Suppose $\mathcal{C}$ is the cutoff instance, resulting in the cutoff transition system $\mathcal{A}^{\mathtt{P}}_{\mathcal{C}} = (\Sigma^{\mathcal{C}}, \Sigma^{\mathcal{C}}_0, \delta_{\mathcal{C}})$. Let $\mathcal{L}$ be some arbitrary protocol instance, resulting in the system $\mathcal{A}^{\mathtt{P}}_{\mathcal{L}} = (\Sigma^{\mathcal{L}}, \Sigma^{\mathcal{L}}_0, \delta_{\mathcal{L}})$. To ensure that $\mathcal{C}$ is a cutoff instance, any trace of $\mathcal{A}^{\mathtt{P}}_{\mathcal{L}}$ which leads to a state violating the safety property should be simulated by a trace of $\mathcal{A}^{\mathtt{P}}_{\mathcal{C}}$ also leading to a state violating the safety property. Consider a relation

$\gamma_{\mathcal{L}} \subseteq \Sigma^{\mathcal{C}} \times \Sigma^{\mathcal{L}}$. We formalize below the conditions which will ensure that $\mathcal{C}$ is a cutoff instance.

$$\varphi_{init}(\gamma_{\mathcal{L}}) \triangleq \forall \sigma_{\mathcal{L}} \in \Sigma_0^{\mathcal{L}}.\ \exists \sigma_{\mathcal{C}} \in \Sigma_0^{\mathcal{C}}.\ (\sigma_{\mathcal{L}}, \sigma_{\mathcal{C}}) \in \gamma_{\mathcal{L}}$$
$$\varphi_{step}(\gamma_{\mathcal{L}}) \triangleq \forall \sigma_{\mathcal{L}}, \sigma_{\mathcal{L}}' \in \Sigma_{\mathcal{L}}.\ \forall \sigma_{\mathcal{C}} \in \Sigma_{\mathcal{C}}.\ \gamma_{\mathcal{L}}(\sigma_{\mathcal{L}}, \sigma_{\mathcal{C}}) \wedge (\sigma_{\mathcal{L}}, a, \sigma_{\mathcal{L}}') \in \delta_{\mathcal{L}}$$
$$\Rightarrow \exists \sigma_{\mathcal{C}}' \in \Sigma_{\mathcal{C}}.\ (\sigma_{\mathcal{C}}, \sigma_{\mathcal{C}}') \in \delta_{\mathcal{C}}^* \wedge \gamma_{\mathcal{L}}(\sigma_{\mathcal{L}}', \sigma_{\mathcal{C}}')$$
$$\varphi_{safety}(\gamma_{\mathcal{L}}) \triangleq \forall \sigma_{\mathcal{L}} \in \Sigma_{\mathcal{L}}.\ \forall \sigma_{\mathcal{C}} \in \Sigma_{\mathcal{C}}.\ \gamma_{\mathcal{L}}(\sigma_{\mathcal{L}}, \sigma_{\mathcal{C}}) \wedge \sigma_{\mathcal{L}} \models \neg\Phi$$
$$\Rightarrow \sigma_{\mathcal{C}} \models \neg\Phi$$

The init condition φ_{init} ensures that every initial state of $\mathcal{A}_{\mathcal{L}}^{\mathtt{P}}$ is related by $\gamma_{\mathcal{L}}$ to some initial state of $\mathcal{A}_{\mathcal{C}}^{\mathtt{P}}$. The step condition φ_{step} ensures that if states of the protocol instance $\mathcal{L}$ and cutoff instance $\mathcal{C}$ are related by $\gamma_{\mathcal{L}}$, then after a transition in $\mathcal{A}_{\mathcal{L}}^{\mathtt{P}}$, the new state of instance $\mathcal{L}$ will continue to be related to a state of $\mathcal{C}$ obtained after 0 or more transitions in $\mathcal{A}_{\mathcal{C}}^{\mathtt{P}}$. Finally, the safety condition φ_{safety} ensures that if a state in $\mathcal{A}_{\mathcal{L}}^{\mathtt{P}}$ violates the safety property (Φ), then its simulating state in $\mathcal{A}_{\mathcal{C}}^{\mathtt{P}}$ also violates the safety property. Together, these conditions ensure that any violating trace of any arbitrary protocol instance can be simulated by a violating trace of the cutoff instance.

Theorem 2. Given a distributed protocol $\mathtt{P}$ and a valid interpretation $\mathcal{C}$, if for any valid interpretation $\mathcal{L}$, there exists a simulation relation $\gamma_{\mathcal{L}}$ such that $(\varphi_{init} \wedge \varphi_{step} \wedge \varphi_{safety})(\gamma_{\mathcal{L}})$, then $\mathcal{C}$ is a cutoff instance of $\mathtt{P}$.

Proof. To show that $\mathcal{C}$ is a cutoff instance, we need to show that if $\mathcal{C}$ is safe, then any protocol instance will be safe. We will show the contra-positive. Consider some protocol instance $\mathcal{L}$ which is not safe. Then, there exists a trace $\tau = \sigma_0^{\mathcal{L}} a_1^{\mathcal{L}} \sigma_1^{\mathcal{L}} \ldots a_n^{\mathcal{L}} \sigma_n^{\mathcal{L}}$ such that $\sigma_n^{\mathcal{L}} \not\models \Phi$, where Φ is the safety property of the protocol. We will construct by induction a trace of the cutoff instance $\tau' = \sigma_0^{\mathcal{C}} a_1^{\mathcal{C}} \sigma_1^{\mathcal{C}} \ldots a_m^{\mathcal{C}} \sigma_m^{\mathcal{C}}$ such that $\sigma_m^{\mathcal{C}} \not\models \Phi$.

Base case: By $\varphi_{init}(\gamma_{\mathcal{L}})$, we know that there exists $\sigma_0^{\mathcal{C}}$ such that $(\sigma_0^{\mathcal{L}}, \sigma_0^{\mathcal{C}}) \in \gamma_{\mathcal{L}}$.

Inductive case: Consider some state $\sigma_i^{\mathcal{L}}$ in the trace τ such that there exists $\sigma_j^{\mathcal{C}}$ in the trace τ' and $(\sigma_i^{\mathcal{L}}, \sigma_j^{\mathcal{C}}) \in \gamma_{\mathcal{L}}$. Since $(\sigma_i^{\mathcal{L}}, a_{i+1}^{\mathcal{L}}, \sigma_{i+1}^{\mathcal{L}}) \in \delta_{\mathcal{L}}$, by $\varphi_{step}(\gamma_{\mathcal{L}})$, there would exists a finite sequence of transitions in $\delta_{\mathcal{C}}$ beginning from $\sigma_j^{\mathcal{C}}$ and ending in some $\sigma_k^{\mathcal{C}}$. We append these transitions to the end of the trace τ' constructed so far.

Hence, at the end, we must have $(\sigma_n^{\mathcal{L}}, \sigma_m^{\mathcal{C}}) \in \gamma_{\mathcal{L}}$. By $\varphi_{safety}(\gamma_{\mathcal{L}})$, we must have $\sigma_m^{\mathcal{C}} \not\models \Phi$, thus proving the result. □

While the above conditions ensure that if the cutoff instance is safe, then any arbitrary protocol instance is also safe, we can further refine them based on the following observation: we only need to simulate till the first violation of the safety property, and hence, we can assume that the safety property holds in all states while simulating till the first violation. The refined step condition φ_{step}^{first} is defined as follows:

$$\varphi_{step}^{first}(\gamma_{\mathcal{L}}) \triangleq \forall \sigma_{\mathcal{L}}, \sigma_{\mathcal{L}}' \in \Sigma_{\mathcal{L}}.\ \forall \sigma_{\mathcal{C}} \in \Sigma_{\mathcal{C}}.\ \gamma_{\mathcal{L}}(\sigma_{\mathcal{L}}, \sigma_{\mathcal{C}}) \wedge (\sigma_{\mathcal{L}}, a, \sigma_{\mathcal{L}}') \in \delta_{\mathcal{L}} \wedge$$
$$\Phi(\sigma_{\mathcal{L}}) \Rightarrow \exists \sigma_{\mathcal{C}}' \in \Sigma_{\mathcal{C}}.\ (\sigma_{\mathcal{C}}, \sigma_{\mathcal{C}}') \in \delta_{\mathcal{C}}^* \wedge \gamma_{\mathcal{L}}(\sigma_{\mathcal{L}}', \sigma_{\mathcal{C}}')$$

Metadata component	Contents
$a \in P.actions$	$a.named\_arguments : list\langle string\rangle$ $a.guard\_atoms : set\langle(x, l, o)\rangle$ $a.body : set\langle(x, l, o)\rangle$ where $x \in functions \cup relations$ $l \in list\langle named\_arguments \cup \{*\}\rangle$ $o \in x.out \cup \{*\}$
$s \in P.sorts$	$string$ that corresponds to a type defined by the protocol
$r \in P.relations$	$r.args : list\langle sorts\rangle$ $r.out = \mathbb{B}$
$f \in P.functions$	$f.args : list\langle sorts\rangle$ $f.out \in sorts$

TABLE I: *The protocol metadata structure P*

Lemma 3. Given a distributed protocol $\mathtt{P}$ and a valid interpretation $\mathcal{C}$, if for any arbitrary valid interpretation $\mathcal{L}$, there exists a simulation relation $\gamma_{\mathcal{L}}$ such that $(\varphi_{init} \wedge \varphi_{step}^{first} \wedge \varphi_{safety})(\gamma_{\mathcal{L}})$, then $\mathcal{C}$ is a cutoff instance of $\mathtt{P}$.

Proof. The proof is the same as the proof for Theorem 2, except that we consider a minimal violating trace of the protocol instance $\mathcal{L}$, such that only the final state in the trace does not satisfy the safety property. □

If the protocol is not safe, then we can consider the first violation of the safety property in any arbitrary instance of the protocol. Since the cutoff instance can simulate this first violation, this would imply that the cutoff instance would also not be safe, thus proving the above lemma. We have found in our experiments that the refined conditions are often more effective in proving *cutoff-ness* of a protocol instance.

V. Synthesizing the Cutoff Instance

In this section, we describe our technique to synthesize the cutoff instance and the simulation relation from the protocol description.

A. Pre-processing & Notation

The protocol description in RML is statically pre-processed to obtain a metadata structure P which has actions, relations, sorts and functions denoted by $P.actions$, $P.sorts$, $P.relations$, $P.functions$. Refer to Table I for a formal description of the protocol metadata structure P and its components. Each action has a set of named arguments, guard atoms and a body. The guard atoms and the function body contain sets of triplets where each triplet contains: the function or relation under consideration, the named arguments of the action (or *) which are its arguments, an output value (or *). The output value indicates constraints that are expected on the relation/function in case of guard atoms; and the updated value of the relation/function entry in case of the body. In all cases, a * represents that the corresponding entry cannot be determined statically and can therefore be unconstrained. As an example, the guard clause $table(n\_old, k, v)$ for the Reshard action would be converted to the triple $(table, [n\_old, k, v], true)$ and the update $seqnum\_sent(s) \leftarrow true$ is converted to the triple $(seqnum\_sent, [s], true)$.

An *instantiation* of an action a is a map from the named arguments of the action to values. A value of $*$ represents that the corresponding named argument can take any value. An *action invocation* is defined as a tuple (a, I) where $a \in P.actions$ and I is an instantiation of a. We define a *clause* as a triple (x, L, o) where $x \in P.relations \cup P.functions$, L is a list of values (some of which can be $*$) conforming to the types in $x.args$ and o is either a constant of type $x.out$ or $*$.

Referring back to our motivating example, an instantiation of the named arguments of the Reshard action would be

$$I = [n\_old : *, n\_new : a_L, k : K, v : *, s : *]$$

and correspondingly, an action invocation would be the tuple $(\text{Reshard}, I)$. Similarly, a clause on the $table$ relation would be $(table, [a_L, K, *], true)$.

B. Static Analysis

Algorithm 4 contains our static analysis algorithm, which takes as input the protocol metadata structure P and an initial set of clauses S_{init}. S_{init} will be derived from the safety property of the protocol; more details are provided in §5.3. S_{init} contains the initial set of clauses relevant for preserving any violation of the safety property. We maintain two sets S and A where S contains a set of clauses and A a set of action invocations. In each iteration, we consider all the new clauses added to the set S in the previous iteration (line 8). For each clause c, in line 9, we invoke ACTIONSTHATSET(P, c) to obtain all the action invocations that potentially set the clause c. We then add the guards for all these action invocations to the set S in line 11. The while loop at line 5 terminates when no new clauses have been added in the previous iteration, thus indicating that we have reached a fixed point.

The function ACTIONSTHATSET(P, c) takes as input the program P and a clause c to return a set of action invocations A which potentially set the clause c. The algorithm works by pattern matching. We iterate over actions and for each atomic update in the body of the action, we check if the atomic update tuple matches the tuple in the clause with respect to the function/relation it updates in line 5. The if condition in line 6 fails only if both the atomic update output and the clause output can be determined statically and they do not match each other.

As an example, assume that the if condition in line 5 passes i.e. both the atomic update and the clause refer to the same function/relation x i.e. $c.x = at\_update.x = x$. If the clause output $c.o = *$ and $at\_update.o = true$ then this means that we are interested in actions that potentially affect $x(c.L)$ in any way, and this atomic update therefore satisfies that requirement. Similarly, if $c.o = true$ and $at\_update.o = *$, this means that we are interested in actions that set $x(c.L) = true$, but the value that the atomic update alters $x(at\_update.l)$ cannot be determined statically. Therefore, conservatively, we assume that the atomic update could potentially alter it as required. But, if $c.o = true$ and $at\_update.o = false$, then the if condition fails as the outputs can be determined statically but do not match.

In line 7 we create an instantiation of $a.named\_arguments$ initialized to *. The PATTERNMATCH function considers the arguments of the update atom and the clause atom $at\_update.l, c.L$ and checks for inconsistencies. For example, $at\_update.l = (a, b, a)$ and $c.L = (1, 2, *)$ would pass the check whereas $at\_update.l = (a, b, a)$ and $c.L = (1, 2, 3)$ would fail the check. If the pattern match succeeds, the for loop instantiaties the named arguments in $at\_update.l$ based on $c.L$. The tuple (a, I) now forms the action invocation which is added to the set of action invocations returned by the algorithm. The GUARDSFOR function returns the set of clauses involved in the guard for an action invocation. We iterate through all the guard atoms of the action in line 3. The for loop in lines 5-6 assigns concrete values to the named arguments in $g.l$ using the instantiation I provided in the action invocation. Then a clause tuple is created in line 7 and added to the list of clauses returned by the algorithm.

As an example of how these methods work, we refer back to sharded key value store example considered in §2. If ACTIONSTHATSET$(P, (unacked, [*, a_L, K, *, *], true))$ is invoked, then one of the actions returned by it would be the Reshard action, with the action invocation $(\text{Reshard}, [n\_old : *, n\_new : a_L, k : K, v : *, s : *])$ (this is because Reshard sets $unacked$ to $true$ in Line-14, Algorithm 1). Similarly, if GUARDSFOR$(\text{Reshard}, [n_old : *, n_new : a_L, k : K, v : *, s : *])$ is invoked, the following set G is returned.

$$G = \{(seqnum\_sent, [*], false), (table, [*, K, *], true)\}$$

C. Synthesizing the Cutoff Instance, Simulation Relation & Lockstep

Cutoff Instance. We start with the safety property Φ in the RML description. As described in §3, the safety property only contains universal quantifiers and hence is a formula of the form $\forall(\bar{x} : \bar{\mathsf{d}}).\ \phi$. The size of the cutoff system is taken to be the number of universally quantified nodes in the safety property.

Obtaining S_{init}. Consider any arbitrary size instance L with $\mathcal{D}_L$ denoting the set of nodes. To begin with the static analysis, we need to provide an initial set of clauses S_{init} as input along with the pre-processed protocol metadata structure P. To obtain S_{init}, we first negate the safety property and instantiate all the existentially quantified variables. We define $\mathcal{D}_L^v \subseteq \mathcal{D}_L$ the set of instantiated nodes or *violating nodes*. We then process the resulting FOL formula $\neg\phi$ to obtain the set of clauses involved in the formula.

As an example, consider the safety property for the Sharded Key Value store protocol from §2. We have

$$\forall N_1, N_2, K, V_1, V_2.\ table(N_1, K, V_1) \wedge table(N_2, K, V_2) \implies N_1 = N_2 \wedge V_1 = V_2$$

As there are 2 quantifiers on nodes, the cutoff for the protocol is 2. Negating and instantiating $N_1 = a_L, N_2 = b_L, K = k, V_1 = v_1$ and $V_2 = v_2$, we get

$$table(a_L, k, v_1) \wedge table(b_L, k, v_2) \wedge (n_1 \neq n_2 \vee v_1 \neq v_2)$$

Algorithm 2 ACTIONSTHATSET

Arguments: P the program, and a clause c

Returns: A a set of action invocations

```
1: procedure ActionsThatSet($P, c$)
2:     $A = \emptyset$
3:     for $a \in P.actions$ do
4:         for $at\_update = (x, l, o)$ in $a.body$ do
5:             if $at\_update.x == c.x$ then
6:                 if $\neg$ ($c.o \neq *$ and $at\_update.o \neq *$ and $c.o \neq at\_update.o$) then
7:                     Create an instantiation $I$ of $a.named\_arguments$, initialized to $*$;
8:                     if PatternMatch($at\_update.l, c.L$) then
9:                         for $i \in 1, len(at\_update.l)$ if $at\_update.l[i] \neq *$ do
10:                            $I[at\_update.l[i]] \leftarrow c.L[i]$
11:                        $r \leftarrow (a, I)$
12:                        $A \leftarrow A \cup \{r\}$
13:    return $A$
```

Algorithm 3 GUARDSFOR

Arguments: P the program, an action invocation act

Returns: G a set of clauses

```
1: procedure GuardsFor($P, act$)
2:     $G = \emptyset$
3:     for $g = (x, l, o) \in a.guards$ do
4:         Create a list $L$ of length $g.l$, initialized to $*$
5:         for $i \in 1, len(g.l)$ if $g.l[i] \neq *$ do
6:             $L[i] \leftarrow act.I[g.l[i]]$
7:         $G \leftarrow G \cup \{(g.x, L, g.o)\}$
8:     return $G$
```

Algorithm 4 STATICANALYSIS

Arguments: P the program, S_{init} a set of clauses

Returns: S a set of clauses, A a set of action invocations

```
1: procedure StaticAnalysis($P, S_{init}$)
2:     $S \leftarrow S_{init}$
3:     $S_{prev} \leftarrow \emptyset$
4:     $A \leftarrow \emptyset$
5:     while $S \neq S_{prev}$ do
6:         $S_d \leftarrow S \setminus S_{prev}$
7:         $S_{prev} \leftarrow S$
8:         for each clause $c$ in $S_d$ do    ▷ For each new clause
9:             $A_t \leftarrow$ ActionsThatSet($P, c$)
10:            for each action invocation $act$ in $A_t$ do
11:                $S \leftarrow S \cup$ GuardsFor($P, act$)
12:            $A \leftarrow A \cup A_t$
13:    return $S, A$
```

giving us the following set of clauses after processing

$$\{(table, [a_L, k, v_1], true), (table, [b_L, k, v_2], true))\}$$

Synthesizing the Simulation Relation and Lockstep. Having obtained S_{init}, we can now invoke STATICANALYSIS(P, S_{init}) to get the set of clauses S and set of action invocations A. We also have the cutoff instance C with its set of nodes $\mathcal{D}_C$. To define the lockstep and simulation relation, we map the nodes of the violating instance to nodes of the cutoff system. Such a mapping $sim : \mathcal{D}_L \rightarrow \mathcal{D}_C$ is defined as follows. Firstly, by construction, $|\mathcal{D}_L^v| = |\mathcal{D}_C|$ i.e., the number of nodes involved in the violation is the same as the number of nodes in the cutoff system. Consequently, we perform a one-to-one mapping of nodes from $\mathcal{D}_L^v$ to $\mathcal{D}_C$. For the rest of the nodes $\mathcal{D}_L \setminus \mathcal{D}_L^v$ in the system L, we make the following observations:

- If S and A obtained from the static analysis do not have any components containing $*$ in any field of the node type, this implies that only actions and state components of the violating nodes are sufficient to simulate the violation. In such a case, there is no need to map nodes from $\mathcal{D}_L \setminus \mathcal{D}_L^v$ as they will never appear in the simulation relation or lockstep.
- If S or A obtained from the static analysis has components containing $*$ in any field of the node type, we map all the nodes from $\mathcal{D}_L \setminus \mathcal{D}_L^v$ to one of the nodes in $\mathcal{D}_C$.

Intuitively, the simulation relation states that for all the clauses that are relevant to the violation (as obtained by the static analysis procedure) in the larger system L, the same state components are maintained in the cutoff system but in the state component of the simulating nodes (as per the sim mapping). Similarly, the lockstep states that the relevant actions are performed in the cutoff system, but by the simulating nodes.

Given S, A and sim, we obtain the simulation relation and lockstep using the procedure SIMANDLOCKSTEP(S, A, sim) in Algorithm 5. The procedure returns the simulation relation γ as a FOL formula and the lockstep τ as an abstract map from action invocations of the larger system to action invocations of the cutoff system. The main idea is to simply perform the relevant actions of A in the cutoff system, whenever they are performed in the larger system, synthesizing the appropriate mapping of the action arguments, and thus maintaining a simulation relation for the relevant state components in S.

Cutoff Verification. To prove that the synthesized cutoff

Algorithm 5 Function to obtain simulation relation and lockstep

Arguments: Set of clauses S, action invocations A and mapping $sim : \mathcal{D}_L \to \mathcal{D}_C$
Returns: FOL formula γ representing the simulation relation and lockstep τ as a map from actions of the larger system to actions of the cutoff system

```
 1: procedure SIMANDLOCKSTEP($S, A, sim$)
 2:     $\gamma \leftarrow true$
 3:     for each clause $c = (x, L, o) \in S$ do
 4:         For each $*$ entry in $L$, replace it with a unique variable name from $\bar{v}$, and add those variables to $L$ to get $\mathcal{L}_{args}$;
 5:         Replace each node variable $n$ in $\mathcal{L}_{args}$ with $sim(n)$ to get $\mathcal{C}_{args}$;
 6:         if $o == *$ then
 7:             ▷ In this case, we assert that the function/relation entries are equal in the larger system and cutoff system
 8:             $\gamma \leftarrow \gamma \bigwedge (\forall \bar{v}.\, x(\mathcal{L}_{args}) = x(\mathcal{C}_{args}))$;
 9:         else
10:             ▷ In this case, we assert that if the relation/function entry takes the value o in L, it also does so in C
11:             if $x.out$ is of node type then
12:                 $\gamma \leftarrow \gamma \bigwedge (\forall \bar{v}.\, (x(\mathcal{L}_{args}) = o) \implies (x(\mathcal{C}_{args}) = sim(o)))$;
13:             else
14:                 $\gamma \leftarrow \gamma \bigwedge (\forall \bar{v}.\, (x(\mathcal{L}_{args}) = o) \implies (x(\mathcal{C}_{args}) = o))$;
15:     Initialize an empty map $\tau$
16:     for each action invocation $act \in A$ do
17:         For each $*$ value in $act.I$, replace it with a unique variable name from $\bar{v}$, to get $act_L.I$;
18:         Replace each node value $n$ in $act_L.I$ with $sim(n)$ to get $act_C.I$;
19:         Define $act_L = (act.a, act_L.I)$ and $act_C = (act.a, act_C.I)$;
20:         $\forall \bar{v}.\, \tau(act_L) \leftarrow act_I$;
21:     return $\gamma, \tau$
```

instance is actually a cutoff for the protocol, we generate FOL formulae for each of the 3 properties $\varphi_{init}(\gamma_{\mathcal{L}}), \varphi_{step}(\gamma_{\mathcal{L}})$ and $\varphi_{safety}(\gamma_{\mathcal{L}})$ mentioned in §3, using the simulation relation γ synthesized by Algorithm 5. Furthermore, for $\varphi_{step}(\gamma_{\mathcal{L}})$, we remove the existential quantifier over the state $\sigma_{\mathcal{C}}^{'}$ after the transition by providing a candidate transition in the system $\mathcal{C}$ as per the lockstep τ.

D. Synthesis for Consensus Protocols

We now describe how the above technique can be adapted to work for quorum-based consensus protocols. Such protocols are used to achieve consensus amongst the nodes on some decision such as proposing a value or choosing a leader, with the safety property being the uniqueness of the decision taken i.e. no two nodes learn of two different decisions.

Quorum-based consensus protocols define a notion of a *quorum* which refers to a set of nodes and a *quorum-set* which is a set of such quorums. Additionally, the quorum-set satisfies the *quorum-intersection* property i.e. any two quorums belonging to a quorum-set intersect. These protocols also involve a voting phase were nodes cast their unique votes for values, and values which receive a quorum of votes are considered as decided. The core safety argument for such protocols typically relies on the quorum-intersection property and the uniqueness of votes i.e. if two values were decided, they both must have received a quorum of votes but since any two quorum-sets intersect, there must be a node that has voted twice which is disallowed by the protocol. Most protocols for achieving consensus such as Raft [12], Paxos [13] and Two-phase commit are designed around these core principles. However, obtaining an inductive invariant for formal verification of these protocols is still a challenging task.

For such protocols, we assume a sort $quorum$ for quorums and a fixed relation $member : node, quorum$ which governs the membership of nodes to quorums. In our pre-processing, for guard atoms in quorum-based consensus protocols, we also track state components on which a quorum-agreement is required.

At a high-level, similar to the non-consensus case, we collect the actions and the state components responsible for a violation through a similar static analysis procedure. However, simulating the violation in the cutoff system by maintaining these states now requires a more complicated lockstep. In particular, the cutoff system tries to maintain the quorum agreement on state components required to reach the violation through staggered actions i.e. the cutoff system waits for a quorum agreement on some necessary state component and then performs the set of actions required to reach quorum agreement in the cutoff system at once thereby ensuring that a quorum agreement on a state component in the larger system is maintained in the cutoff system.

We illustrate this on a toy consensus protocol [14]. The goal of the protocol is for the nodes involved to decide on a single value. Nodes *vote* for values and a value is *decided* once a *quorum* of nodes have voted for the same value. Refer to Fig. 6 for a detailed pseudocode description of the protocol.

A sort (or type) is defined for nodes, values and quorums. Additionally, the protocol description also contains an *axioms* on the $member$ relation (which marks the membership of nodes to quorums) that formally specifies the quorum-intersection property. The $\mathsf{CastVote}(n, v)$ action encapsulates the semantics of a node voting for a value. The guard ensures that the node has not $voted$ in the past and the body of the action marks the node to have voted and updates the $vote$ table to indicate that node n has voted for value v. The Decide(v) action requires the existence of a quorum of nodes who have all voted for the value v, in which case, the value v is marked as decided.

The correctness of the toy consensus protocol depends on an intricate interplay between the quorum-intersection property and the fact that the protocol does not allow nodes to vote for two different values. Intuitively, if two distinct values are *decided*, then there must be a quorum of nodes who agree on each of the two values. From the quorum-intersection property, these two quorums have at-least one node in common which must therefore have voted twice which is disallowed by the protocol. An inductive invariant for this protocol would have to explicitly encode this logic and disallow any and all states that might potentially lead to state that has two decided values.

Algorithm 6 The toy consensus protocol in RML

```
1: type node, value, quorum
2: relation vote: node, value
3: relation voted: node
4: relation member: node, quorum
5: relation decided: value
6: axiom $\forall q_1.q_2.\exists n.\ member(n, q_1) \wedge member(n, q_2)$
7: init:
8:     $vote(*, *) \leftarrow false$
9:     $voted(*) \leftarrow false$
10:    $decided(*) \leftarrow false$
11: action CastVote($n : node, v : value$)
12:    require $\neg voted(n)$
13:    $vote(n, v) \leftarrow true$
14:    $voted(n) \leftarrow true$
15: action Decide($v : value, q : quorum$)
16:    require $\forall N.member(N, q) \implies vote(N, v)$
17:    $decided(v) \leftarrow true$
18: safety $\forall v_1.v_2.\ decided(v_1) \wedge decided(v_2) \implies v_1 = v_2$
```

We show that the toy consensus protocol has a cutoff of 1 i.e. a single node system can encapsulate all the possible ways in which this protocol might violate the safety property.

Our simulation relation ensures the following, if a quorum of nodes have voted for a value $v \in \{v_1, v_2\}$ in the larger system, a quorum of nodes also vote for v in the cutoff system. This is maintained through a staggered lockstep i.e. the cutoff system remains in the same state as long as an action in the larger system has not triggered any quorum of $vote$'s. Once a quorum of nodes vote for a value v in the larger system, we trigger the CastVote($*, v$) action on a quorum of nodes (only 1 node) in the cutoff system. To ensure that this action can be fired, we also add a 'clean state' clause that maintains the following property, if there does not exist a quorum of nodes that have voted for either v_1 or v_2, then the cutoff system is still in an initial state. This property is clearly maintained by virtue of how we design the lockstep.

Such an analysis can be extended for arbitrarily complex consensus protocols such as the Two-Phase Commit, Ricart-Agrawala and the generic consensus protocol modelled in [14].

Protocol	**Cutoff**	**Time Taken(s)**	$\lvert\gamma\rvert$
Sharded Key-Value Store[16]	2	0.02	5
Leader Election in a Ring[17]	2	0.03	4
Centralized Lock Server[18]	2	0.02	5
Lock Server Sync[14]	2	0.01	2
Ricart Agrawala[19]	2	0.01	6
Two Phase Commit[20]	2	0.02	9
Toy Consensus ForAll[14]	1	0.07	5
Consensus[14]	2	29.7	11

TABLE II: *γ is a FOL formula of the type $\bigwedge_{i=1}^{|\gamma|}(p \implies q)$ therefore $|\gamma|$ represents the number of clauses of the type $p \implies q$ in the simulation relation. Time taken refers to the total time taken by our synthesis+verification procedure.*

VI. Experimental Results

We have applied the proposed strategy on a variety of different distributed protocols given in Table II. Our technique works in two parts, where we first attempt to automatically synthesize the cutoff instance, and then attempt to prove its correctness. For proving correctness of a cutoff instance, we generate a FOL encoding of the 3 conditions $\varphi_{init}(\gamma_{\mathcal{L}}), \varphi_{step}(\gamma_{\mathcal{L}}, \tau_{\mathcal{L}})$ and $\varphi_{safety}(\gamma_{\mathcal{L}})$. We reduce the problem of checking correctness to satisfiability of the generated FOL formulae. For example, for checking the $\varphi_{step}(\gamma_{\mathcal{L}}, \tau_{\mathcal{L}})$ condition which is a condition of the type $p \implies q$ to be correct, we check whether $p \wedge \neg q$ is unsatisfiable. We use Z3 [15] as our backend SMT solver. The experiments were run on a system with a 12-core Apple M2 Pro processor and 16GB RAM. Table II summarizes our experimental results. Notice that the time taken for each protocol is in the order of few milliseconds except for the Consensus protocol which takes significantly longer due to the larger number of quantifiers used in the encoding.

We now present a detailed description of each protocol and its cutoff instance

Leader Election in a Ring. We considered the Leader Election in a ring protocol as presented in [17]. The protocol deals with electing a unique leader in a ring setting. Each node has a unique ID and there exists a total order on the IDs of the nodes. The protocol defines static relations $between$ (to define a notion of a node being between two nodes) and $next$ (to define the clockwise neighbouring node for every node) which are used to describe the ring topology. The protocol works as follows, each node sends a message containing its ID to its clockwise neighbour. When any node receives an ID, the ID is forwarded to its neighbour only if the ID in the message is greater than its own. When a node receives its own ID, it elects itself as a $leader$. The safety property describes that no two nodes are ever elected as leaders i.e. $leader(N_1) \wedge leader(N_2) \implies false$. Intuitively, the protocol works correctly because we have a total order on the IDs and therefore only the node with the highest ID is elected as the leader.

To synthesize a cutoff instance, we instantiate a violation in an arbitrary instance containing two nodes a_L and b_L such that $leader(a_L)$ and $leader(b_L)$ both hold. We then synthesize a cutoff instance with 2 nodes a_C and b_C such that $ID(a_L) = ID(a_C)$ and $ID(b_L) = ID(b_C)$. With the sim relation mapping nodes between a_L and b_L (in the direction of the ring) to b_C and nodes between b_L and a_L to a_C, along with the S and A sets obtained from the lockstep, we are able to generate a correct cutoff instance.

Centralized Lock Server. The Lock Server protocol [18], implements a centralized lock service. Clients request for the lock by sending a $lock$ message. Similarly, a client can relinquish the lock (if it owns it) by sending an $unlock$ message. The protocol allows for the client to send arbitrary number of duplicate $lock$ messages. The network picks a message at random to be handled by the server. The server maintains a queue of client nodes that have requested for the lock while granting the lock the node at the head of the queue. To handle an $unlock$ message, the server removes the node from the head of the queue and sends a $grant$ message to the new head (if any). On receiving a $grant$ message, the receiving client node holds the lock. The safety property is again mutual exclusion i.e. no two nodes hold the lock simultaneously.

We implemented the protocol in RML by emulating the network semantics. We implement a unique sequence number for every $lock$, $unlock$ and $grant$ message. The functionality of picking a random network message is then equivalent to firing a $handle$ action for one of the previously un-handled sequence numbers at random. We also implement the queue as a triple $(head, tail, q)$ where $head$ and $tail$ are integers and $q : \mathbb{Z} \rightarrow node$ is a function that outputs the queue elements for valid indices between $head$ and $tail$ inclusive.

To synthesize the cutoff instance, we instantiate a violation with two nodes a_L and b_L that both hold the lock. In the cutoff system, we have two nodes a_C and b_C. The static analysis requires us to simulate the actions of nodes apart from the violating nodes, hence we map $sim(a_L) = a_C$ and the rest of the nodes to b_C. At a high level, the $lock, unlock$ and $grant$ messages of all nodes that map to b_C are sent and handled by the node b_C in the cutoff system. The simulation relation also states that the queue in the cutoff system is identical to the queue of the larger system with the sim mapping applied to the contents of the queue.

Lock Server Sync. This protocol [14] implements a synchronous lock server protocol. Sorts are defined for clients and servers. A client c connects to a server s through the Connect action which requires the server to own the semaphore/lock. This creates a $link$ from c to s and the servers ownership of the lock is revoked. For a client c to disconnect (as per the Disconnect action) from a server, we require the existence of a $link$ from c to s. The action disconnects the link and grants the ownership of the lock back to the server. The safety property says that at all points of time, only one client can have a link with a given server i.e. $link(c_1, s) \wedge link(c_2, s) \implies c_1 = c_2$.

We provide a cutoff of 2 on the client sort i.e. a system with 2 clients can exhibit all possible ways in which the safety property might be violated. We instantiate a violation with two clients a_L, b_L and a server S, to obtain $S_{init} = \{(link, (a_L, S), true), link(b_L, S), true)\}$. Consequently, the cutoff system contains two nodes a_C, b_C and a server S. Proceeding with the static analysis algorithm and using a sim mapping that maps $a_L \rightarrow a_C$ and the rest of the nodes to b_C gives us the required lockstep and simulation relation to establish cutoff of 2.

Ricart-Agrawala. We consider the Ricart Agrawala protocol for distributed mutual exclusion as presented in [19]. A node sends request messages to all other nodes to enter the critical section. A responder node replies to a requester only if the responder is not holding the lock and hasn't received nor sent a reply to/from the requester earlier. To enter the critical section, a node must have received a reply from every other node. To leave the critical section, a node simply discards previous replies and relinquishes control of the lock. The safety property is mutual exclusion i.e. no two nodes will ever enter the critical section/hold the lock simultaneously.

In synthesizing the cutoff instance, we start with a violation in an arbitrary instance where two nodes a_L and b_L held the lock simultaneously. The resulting static analysis yielded sets A and S where only the actions of the two violating nodes and their state components were relevant to simulate the violation. We thus simulate a cutoff instance with two nodes a_C and b_C with $sim(a_L) = a_C$ and $sim(b_L) = b_C$. Note that there is no need to simulate actions of other nodes as they do not appear in the backward analysis. This is noteworthy, as for other protocols, the backward analysis eventually results in inclusion of actions and state components of non-violating nodes as well.

Two Phase Commit: We consider the Two-Phase Commit protocol as provided in [5], [20] for performing atomic commits in a distributed setting. The original protocol allows nodes to Vote for one of $true/false$ that signifies a complete/abort locally, and also allows nodes to fail arbitrarily. We look at a simplified version of the protocol where nodes cannot fail. The global actions Go1 is triggered if all the nodes have voted $true$ (commit), whereas the Go2 action is triggered if there exists a node that has voted $false$ (abort). After the global actions Go1 or Go2 are triggered, the nodes can locally $decide\_commit$ or $decide\_abort$ respectively. We consider the key safety property that states that $decide\_commit(N_1) \wedge decide\_abort(N_2) \implies false$ i.e. we cannot have two nodes where one decides to commit and the other aborts.

We can model this protocol as a quorum-based consensus protocols by generalizing the notion of a quorum. Specifically, we can consider a vote commit quorum $q_1 \in Q_1$ to be a set consisting of all the nodes in the system (the quorum-set Q_1 is therefore singleton), and a vote abort quorum $q_0 \in Q_0$ to be a set containing any single node (the quorum-set Q_0 therefore contains one set for each of the nodes in the system). Similar to quorum-based consensus protocols, we cannot have two different nodes a_L and b_L with $decide\_commit(a_L)$ and $decide\_abort(b_L)$, because the two quorums required for these decisions q_1 and q_0 always intersect which would require the existence of a node has voted for commit and abort.

Through our technique, we obtain a cutoff instance with two nodes a_C and b_C. The simulation relation ensures that if all the nodes in the larger system have voted for commit, then both a_C and b_C vote for commit in the cutoff system. This is maintained through staggered vote commit actions that are triggered in the cutoff system only when all the nodes in the larger system have voted yes. Similarly, by staggering vote abort actions, we ensure that if there exists a node that has voted to abort in the larger system, such a node also exists in the cutoff system. Lastly, we also have a 'clean state' clause that allows for staggered actions, which ensures that if neither of the quorum-events have occurred in the larger system i.e. neither have all the nodes voted yes, nor does there exist a node that has voted no; then the cutoff system is in a clean state without any entries in its relation tables.

Consensus: The consensus protocol [14] implements a generic quorum-based consensus protocol in a fully asynchronous manner with nodes requesting for votes, receiving votes followed by a decide action to pick the leader based on a quorum of votes received. Despite being a significantly complicated protocol, our approach successfully shows a cutoff of 2 for this protocol

VII. Related Work and Conclusion

In the recent past, there has been a lot of interest in automated and mechanised verification of distributed protocols ([1]–[6]). Ironfleet [16] and Verdi [18] are some of the earliest works which are more focused towards verifying real-world implementations of distributed protocols, and typically assume that an abstract model of the protocol works correctly. Many of the recent approaches towards protocol verification rely on constructing and proving some form of inductive invariant. Padon et. al. [11] introduced the Ivy framework along with the RML language which allows a protocol developer to interactively generate an inductive invariant for verifying safety. Other approaches ([1], [2], [5]) have continued along this line of work, by attempting to automate the process of deriving the inductive invariant using techniques like IC3/PDR or data-driven approaches. While these approaches have been successful to some extent, we note that the problem of deriving inductive invariants is a fundamentally hard problem, and our work allows us to sidestep it. In fact, it could be useful to apply these techniques to the comparatively simpler problem of finding and proving a cutoff instance.

While previous works have also attempted to use cutoff-based approaches for verification ([7]–[10]), they have mostly been limited to either a restricted class of protocols or a restricted class of specifications. We note that none of these works actually mechanize and automate the proof that a protocol instance is actually a cutoff instance. To our best knowledge, ours is the first work that enables automated cutoff based verification.

In this work, we investigated the applicability of cutoff based verification for a variety of distributed protocols. We observe that cutoff based verification allows us to naturally sidestep the harder problem of finding inductive invariants. We identify sufficient conditions which can be used to verify that a protocol instance is indeed a cutoff instance and which can be encoded using SMT. We develop a simple static analysis-based approach to automatically synthesize the cutoff instance for many protocols.

We note that our approach has limitations. In particular, it can primarily fail in one of two ways. Firstly, the cutoff value itself could be higher than the one chosen by our analysis. Secondly, it is possible that the simulation relation and the lockstep synthesized by our analysis may not work (i.e. they may not satisfy the φ_{step} or φ_{safety} constraints). In either case, our analysis will not succeed in verifying the protocol. Intuitively, this could happen because the nodes in our synthesized cutoff instance cannot simulate a violation of the safety property, in which case, either of the φ constraints will not hold. One can construct an artificial example to demonstrate this; however, we note that we have not encountered this issue in our experiments. It is a well-established empirical result that most bugs in real-world protocol implementations and designs can be discovered within a small scope of parameter values. Our work takes a step towards generalizing and formalizing this result by providing a generic simulation-based strategy to synthesize cutoff instances and cutoff proofs.

To conclude, our cutoff-based verification approach demonstrates how a combination of static analysis, SMT-based verification, and model checking can simplify the hard problem of protocol verification. Our experimental results indicate that cutoff results are ubiquitous and applicable for different types of protocols. Our vision is that this work can pave the way for more investigations into automating cutoff results for more complex protocols.

References

[1] Y. M. Y. Feldman, J. R. Wilcox, S. Shoham, and M. Sagiv, "Inferring inductive invariants from phase structures," in *CAV (2)*, ser. Lecture Notes in Computer Science, vol. 11562. Springer, 2019, pp. 405–425.

[2] H. Ma, A. Goel, J. Jeannin, M. Kapritsos, B. Kasikci, and K. A. Sakallah, "I4: incremental inference of inductive invariants for verification of distributed protocols," in *SOSP*. ACM, 2019, pp. 370–384.

[3] K. L. McMillan and O. Padon, "Ivy: A multi-modal verification tool for distributed algorithms," in *CAV (2)*, ser. Lecture Notes in Computer Science, vol. 12225. Springer, 2020, pp. 190–202.

[4] O. Padon, G. Losa, M. Sagiv, and S. Shoham, "Paxos made EPR: decidable reasoning about distributed protocols," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, pp. 108:1–108:31, 2017.

[5] J. Yao, R. Tao, R. Gu, J. Nieh, S. Jana, and G. Ryan, "Distai: Data-driven automated invariant learning for distributed protocols," in *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*, A. D. Brown and J. R. Lorch, Eds. USENIX Association, 2021, pp. 405–421.

[6] A. Damian, C. Dragoi, A. Militaru, and J. Widder, "Communication-closed asynchronous protocols," in *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part II*, ser. Lecture Notes in Computer Science, I. Dillig and S. Tasiran, Eds., vol. 11562. Springer, 2019, pp. 344–363. [Online]. Available: https://doi.org/10.1007/978-3-030-25543-5_20

[7] E. A. Emerson and K. S. Namjoshi, "Reasoning about rings," in *POPL*. ACM Press, 1995, pp. 85–94.

[8] N. Jaber, S. Jacobs, C. Wagner, M. Kulkarni, and R. Samanta, "Parameterized verification of systems with global synchronization and guards," in *CAV (1)*, ser. Lecture Notes in Computer Science, vol. 12224. Springer, 2020, pp. 299–323.

[9] O. Maric, C. Sprenger, and D. A. Basin, "Cutoff bounds for consensus algorithms," in *CAV (2)*, ser. Lecture Notes in Computer Science, vol. 10427. Springer, 2017, pp. 217–237.

[10] R. Bloem, S. Jacobs, A. Khalimov, I. Konnov, S. Rubin, H. Veith, and J. Widder, *Decidability of Parameterized Verification*, ser. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2015.

[11] O. Padon, K. L. McMillan, A. Panda, M. Sagiv, and S. Shoham, "Ivy: safety verification by interactive generalization," in *PLDI*. ACM, 2016, pp. 614–630.

[12] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. Philadelphia, PA: USENIX Association, Jun. 2014, pp. 305–319.

[13] L. Lamport, "The part-time parliament," *ACM Trans. Comput. Syst.*, vol. 16, no. 2, p. 133–169, may 1998.

[14] J. Yao, R. Tao, R. Gu, and J. Nieh, "DuoAI: Fast, automated inference of inductive invariants for verifying distributed protocols," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, Carlsbad, CA, Jul. 2022, pp. 485–501.

[15] L. M. de Moura and N. Bjørner, "Z3: an efficient SMT solver," in *TACAS*, ser. Lecture Notes in Computer Science, vol. 4963. Springer, 2008, pp. 337–340.

[16] C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. T. V. Setty, and B. Zill, "Ironfleet: proving practical distributed systems correct," in *SOSP*. ACM, 2015, pp. 1–17.

[17] E. Chang and R. Roberts, "An improved algorithm for decentralized extrema-finding in circular configurations of processes," *Commun. ACM*, vol. 22, no. 5, p. 281–283, may 1979.

[18] J. R. Wilcox, D. Woos, P. Panchekha, Z. Tatlock, X. Wang, M. D. Ernst, and T. E. Anderson, "Verdi: a framework for implementing and formally verifying distributed systems," in *PLDI*. ACM, 2015, pp. 357–368.

[19] G. Ricart and A. K. Agrawala, "An optimal algorithm for mutual exclusion in computer networks," *Commun. ACM*, vol. 24, no. 1, p. 9–17, jan 1981.

[20] J. N. Gray, *Notes on data base operating systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1978, pp. 393–481.

[21] K. S. Namjoshi, "Symmetry and completeness in the analysis of parameterized systems," in *VMCAI*, ser. Lecture Notes in Computer Science, vol. 4349. Springer, 2007, pp. 299–313.

[22] M. Taube, G. Losa, K. L. McMillan, O. Padon, M. Sagiv, S. Shoham, J. R. Wilcox, and D. Woos, "Modularity for decidability of deductive verification with applications to distributed systems," in *PLDI*. ACM, 2018, pp. 662–677.

[23] S. Chand, Y. A. Liu, and S. D. Stoller, "Formal verification of multi-paxos for distributed consensus," in *International Symposium on Formal Methods*. Springer, 2016, pp. 119–136.

[24] V. Rahli, D. Guaspari, M. Bickford, and R. L. Constable, "Formal specification, verification, and implementation of fault-tolerant systems using eventml," *Electron. Commun. Eur. Assoc. Softw. Sci. Technol.*, vol. 72, 2015. [Online]. Available: https://api.semanticscholar.org/CorpusID:46662559

[25] S. Paul, G. A. Agha, S. Patterson, and C. A. Varela, "Verification of eventual consensus in synod using a failure-aware actor model," in *NASA Formal Methods Symposium*. Springer, 2021, pp. 249–267.

[26] P. Küfner, U. Nestmann, and C. Rickmann, "Formal verification of distributed algorithms," in *Theoretical Computer Science*, J. C. M. Baeten, T. Ball, and F. S. de Boer, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 209–224.

[27] B. Charron-Bost and A. Schiper, "Schiper, a.: The heard-of model: computing in distributed systems with benign faults. distributed computing 22(1), 49-71," *Distributed Computing*, vol. 22, 04 2009.

[28] K. Chaudhuri, D. Doligez, L. Lamport, and S. Merz, "Verifying safety properties with the tla+ proof system," in *Automated Reasoning*, J. Giesl and R. Hähnle, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 142–148.