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 together.

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 (11–61) 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.

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 that describes the state of the protocol and are defined over these sorts. The relations

![Algorithm 1 The Sharded Key Value Store Protocol](image)

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.
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 acknowledgement. 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 pairs 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 $table(A_L(B_L), K, V_1(V_2))$ (we use entries in brackets $\langle \rangle$ to succinctly represent both the clauses). We find that any action of the type $\text{Put}(A_L(B_L), K, V_1(V_2))$ and $\text{RecvTransferMsg}(*, A_L(B_L), K, V_1(V_2), *)$ 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 $\text{RecvTransferMsg}$ actions, the guard contains the clauses $\neg seqnum_recv(*)$ and $\text{transfer_msg}(*, A_L(B_L), K, V_1(V_2), *)$. For the Put actions, we have $\exists v. \text{table}(A_L(B_L), K, v)$ as the guard clause. For the existential quantifier, we include $\text{table}(A_L(B_L), K, *)$ 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 $\text{table}(A_L, K, V_1)$ and also an entry $\text{table}(A_L, K, *)$, the latter subsumes the former. On performing such reductions, we get the following fixed point sets $S$ and $A$

$$S = \{\text{table}(*, K, *), \text{transfer_msg}(*, *, K, *, *), \neg seqnum_recv(*), \neg seqnum_sent(*), \text{unacked}(*, *, K, *, *)\}$$

$$A = \{\text{Put}(*, K, *), \text{RecvTransferMsg}(*, *, K, *, *), \text{Reshard}(*, *, K, *, *), \text{Retransmit}(*, *, K, *, *)\}$$

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 $\text{DropTransferMsg}$ and $\text{SendAck}$ 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 $sim : D_L \rightarrow D_C$ (where $D_x$ 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 $n \in D_L$ 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. $\forall N \in D_L, (N \neq A) \land (N \neq B) \implies sim(N) = B_C$. Intuitively, a node $N_C \in D_C$ maintains the state and performs the actions for all the nodes $N_L \in D_L$ 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. Put$_L(n, K, c)$ is simulated as Put$_C(sim(N), K, V)$
2. Reshard$_L(n_1, n_2, K, v, s)$ is simulated as Reshard$_C(sim(n_1), sim(n_2), K, v, s)$
3. Retransmit$_L(n_1, n_2, K, v, s)$ is simulated as Retransmit$_C(sim(n_1), sim(n_2), K, v, s)$
4. RecvTransferMsg$_L(n_1, n_2, K, v, s)$ is simulated as RecvTransferMsg$_C(sim(n_1), sim(n_2), K, v, s)$

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 $P = (D, R, F, \Psi, \Lambda, \Phi)$ consists of a set of declarations $(D, R, F)$, axioms ($\Psi$), actions ($\Lambda$) and a safety property ($\Phi$). The declarations define the vocabulary: $D$, $R$ and $F$ denote the set of domain names,
relation names and function names respectively (along with the relation and function signatures). The axioms (Ψ) are FOL formulæ defined over the vocabulary which encode properties of the domains. Φ denotes the safety property, which is another FOL formula, while 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 $A^p_\Sigma = (\Sigma, \Sigma_0, \delta)$ is parameterized by a domain interpretation function $\mathcal{I}$ which associates a finite domain of values with each domain name $d \in D$. For the interpretation function $\mathcal{I}$ to be valid, we require the domains in range of $\mathcal{I}$ to satisfy all the axioms in $\Psi$. Each state $\sigma \in \Sigma$ is an interpretation of function and relation names in $F$ and $R$ to actual functions and relations over the domains defined by the interpretation function $\mathcal{I}$. That is, for a function signature $\ell : (d_1 \times \ldots \times d_n) \rightarrow d$ in the protocol description, $\sigma(\ell)$ will be a function of the form $\mathcal{I}(d_1) \times \ldots \mathcal{I}(d_n) \rightarrow \mathcal{I}(d)$. The same holds for a relation $r$ in the description.

The RML protocol description also consists of a set of axioms $\Psi_0$ 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 ($\sigma$) being the model of an FOL formula ($\Psi$).

Transitions of $A^p_\Sigma$ will correspond to actions of the protocol. An action $a(\bar{v} : \bar{d}) : (g(\bar{v}), u(\bar{v}))$ is parameterized over a set of (typed) variable names $\bar{v}$, and consists of two components: (i) an FOL formula $g$ (also called the guard) which can contain free variables from $\bar{v}$, (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 $A^p_\Sigma$ caused by the action $a$ in the protocol are formally defined as follows:

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

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

The safety property $\Phi$ is defined as a FOL formulæ using the declared domains, functions and relations. In this work, we assume that $\Phi$ only uses universal quantifiers. Hence, $\Phi$ has the form : $\forall(\bar{x} : \bar{d}), \phi$. 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 $A^p_\Sigma$ is a sequence of states and transition labels of the form $\sigma_0 \sigma_1 \sigma_2 \ldots \sigma_n$ 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 $\mathcal{T}(A^p_\Sigma)$ denote the set of traces of $A^p_\Sigma$. We use $[A^p_\Sigma]$ to denote the set of reachable states of $A^p_\Sigma$, i.e. $[A^p_\Sigma] = \{ \sigma^t \mid \sigma_0 \ldots \sigma^t \in \mathcal{T}(A^p_\Sigma) \}$. A transition system is safe if all of reachable states obey the safety property of the protocol:

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

While $A^p_\mathcal{I}$ 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 $\Psi$ 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 $P = (D, R, F, \Psi, \Phi, A)$ is safe if for every valid domain interpretation function $\mathcal{I}$ satisfying the axioms $\Psi$, $A^p_\mathcal{I}$ 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 $P$, a cutoff instance $C$ is a valid interpretation of domains such that if $A^p_C$ is safe, then for any valid interpretation $\mathcal{I}$, $A^p_\mathcal{I}$ is safe.

**Theorem 1.** For a distributed protocol $P$, if $C$ is a cutoff instance, and $A^p_C$ is safe, then the distributed protocol $P$ is safe.1

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 $C$ is the cutoff instance, resulting in the cutoff transition system $A^p_C = (\Sigma^C, \Sigma^C_0, \delta_C)$. Let $L$ be an arbitrary protocol instance, resulting in the system $A^p_L = (\Sigma^L, \Sigma^L_0, \delta_L)$. To ensure that $C$ is a cutoff instance, any trace of $A^p_L$ which leads to a state violating the safety property should be simulated by a trace of $A^p_C$ also leading to a state violating the safety property. Consider a relation

1The proofs for the theorems are provided in the full version of our paper at https://github.com/shreesha00/FMCAD.git
\( \gamma_C \subseteq \Sigma^C \times \Sigma^C \). We formalize below the conditions which will ensure that \( C \) is a cutoff instance.

\[
\begin{align*}
\varphi_{\text{init}}(\gamma_C) & \equiv \forall \sigma_C \in \Sigma^C, \exists \sigma_C \in \Sigma^C. (\sigma_C, \sigma_C) \in \gamma_C \\
\varphi_{\text{step}}(\gamma_C) & \equiv \forall \sigma_C, \sigma_C' \in \Sigma^C, \exists \sigma_C \in \Sigma^C. \gamma_C(\sigma_C, \sigma_C') \land (\sigma_C, a, a') \in \delta_C \\
& \Rightarrow \exists \sigma_C' \in \Sigma^C. (\sigma_C, \sigma_C') \in \delta^C \land \gamma_C(\sigma_C', \sigma_C') \\
\varphi_{\text{safety}}(\gamma_C) & \equiv \forall \sigma_C \in \Sigma^C. \forall \sigma_C \in \Sigma^C. \gamma_C(\sigma_C, \sigma_C) \land \in \delta_C \\
& \Rightarrow \sigma_C \models \neg \Phi
\end{align*}
\]

The init condition \( \varphi_{\text{init}} \) ensures that every initial state of \( A_C^P \) is related by \( \gamma_C \) to some initial state of \( A_C^P \). The step condition \( \varphi_{\text{step}} \) ensures that if states of the protocol instance \( L_C^P \) and cutoff instance \( C \) are related by \( \gamma_C \), then after a transition in \( A_C^P \), the new state of instance \( C \) will continue to be related to a state of \( C \) obtained after 0 or more transitions in \( A_C^P \). Finally, the safety condition \( \varphi_{\text{safety}} \) ensures that if a state in \( A_C^P \) violates the safety property \( \Phi \), then its simulating state in \( A_C^P \) 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 \( P \) and a valid interpretation \( C \), if for any valid interpretation \( L \), there exists a simulation relation \( \gamma_C \) such that \( (\varphi_{\text{init}} \land \varphi_{\text{step}} \land \varphi_{\text{safety}})(\gamma_C) \), then \( C \) is a cutoff instance of \( P \).

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 \( \varphi_{\text{first}} \) is defined as follows:

\[
\begin{align*}
\varphi_{\text{first}}(\gamma_C) & \equiv \forall \sigma_C, \sigma_C' \in \Sigma^C. \forall \sigma_C \in \Sigma^C. \gamma_C(\sigma_C, \sigma_C') \land (\sigma_C, a, a') \in \delta_C \land \\
& \Phi(\sigma_C) \Rightarrow \exists \sigma_C' \in \Sigma^C. (\sigma_C, \sigma_C') \in \delta^C \land \gamma_C(\sigma_C', \sigma_C')
\end{align*}
\]

**Lemma 3.** Given a distributed protocol \( P \) and a valid interpretation \( C \), if for any arbitrary valid interpretation \( L \), there exists a simulation relation \( \gamma_C \) such that \( (\varphi_{\text{first}} \land \varphi_{\text{safety}})(\gamma_C) \), then \( C \) is a cutoff instance of \( P \).

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 be safe; thus, proving the above lemma. We have found in our experiments that the refined conditions are often more effective in proving cutoffness 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.

---

**TABLE I: The protocol metadata structure \( P \)**

<table>
<thead>
<tr>
<th>Metadata component</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a \in \text{P.actions} )</td>
<td>( a.text )</td>
</tr>
<tr>
<td>( a \in \text{P.actions} )</td>
<td>( a.guard )</td>
</tr>
<tr>
<td>( a \in \text{P.actions} )</td>
<td>( a.body )</td>
</tr>
<tr>
<td>( s \in \text{P.sorts} )</td>
<td>( s )</td>
</tr>
<tr>
<td>( r \in \text{P.relations} )</td>
<td>( r )</td>
</tr>
<tr>
<td>( f \in \text{P.functions} )</td>
<td>( f )</td>
</tr>
</tbody>
</table>

---

**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 \( P\text{-actions} \), \( P\text{-sorts} \), \( P\text{-relations} \), \( P\text{-functions} \). 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 atom 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 \( \text{table}(n\_old, k, v) \) for the Reshard action would be converted to the triple \( (\text{table}, [n\_old, k, v], \text{true}) \) and the update \( \text{seqnum\_sent}(s) \leftarrow \text{true} \) is converted to the triple \( (\text{seqnum\_sent}, [s], \text{true}) \).

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 \( a \in \text{P.actions} \) and \( I \) is an instantiation of \( a \). We define a clause as a triple \( (x, L, o) \) where \( x \in \text{P.relations} \cup \text{P.functions} \), \( L \) is a list of values (some of which can be \( * \)) conforming to the types in \( x\text{-args} \) and \( o \) is either a constant of type \( x\text{-out} \) or \( * \).

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

\[
\]

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

---

**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 \( S_{\text{init}} \). \( S_{\text{init}} \) will be derived from the safety property of the protocol; more details are provided in §5.3. \( S_{\text{init}} \) 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. If the 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 matches 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. If the 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 matches 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. If the 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 matches 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. If the 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 matches 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. If the 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 matches 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. If the 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.
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 $\neg (c.o \neq \ast \mbox{ and } at\_update.o \neq \ast \mbox{ and } c.o \neq at\_update.o)$ then
7:                  Create an instantiation $I$ of $a$\_.named\_arguments, initialized to $\ast$;
8:              if $\neg$ $\mbox{PATTERNMATCH}(at\_update.l, c.L)$ then
9:                  for $i \in 1, len(at\_update.l)$ if $at\_update.l[i] \neq \ast$ do
10:                     $I[\ast] \leftarrow c.L[i]$
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 $\ast$
5:      for $i \in 1, len(g.l)$ if $g.l[i] \neq \ast$ 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 $S, A$

From $D_L \setminus D_L^v$ as they will never appear in the simulation relation or lockstep.

- If $S$ or $A$ obtained from the static analysis has components containing $\ast$ in any field of the node type, we map all the nodes from $D_L \setminus D_L^v$ to one of the nodes in $D_C$.

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

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 where nodes cast their unique votes for values, and values
Algorithm 5 Function to obtain simulation relation and lockstep

Arguments: Set of clauses $S$, action invocations $A$ and mapping sim : $D_L → D_C$

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($S, A, \text{sim}$)
2:  $γ \leftarrow \text{true}$
3:  for each clause $c = (x, L, o) \in S$
4:    For each * entry in $L$, replace it with a unique variable name from $\bar{v}$, and add those variables to $L$ to get $L_{args}$;
5:    Replace each node variable $n$ in $L_{args}$ with sim($n$) to get $C_{args}$;
6:    if $a == *$ then
7:      $γ \leftarrow γ \land (\forall \bar{v}. x(L_{args}) = x(C_{args}))$;
8:    else
9:      $γ \leftarrow γ \land (\forall \bar{v}. (x(L_{args}) = o) \implies (x(C_{args}) = \text{sim}(o)))$;
10:  end if
11: if $x$.out is of node type then
12:    $γ \leftarrow γ \land (\forall \bar{v}. (x(L_{args}) = o) \implies (x(C_{args}) = o)))$;
13: else
14:    $γ \leftarrow γ \land (\forall \bar{v}. (x(L_{args}) = o) \implies (x(C_{args}) = o)))$;
15:  end if
16: end for
17: for each action invocation $act \in A$
18:    For each * value in act.$I$, replace it with a unique variable name from $\bar{v}$, to get $act_L$.
19:    Replace each node value $n$ in act$_L$ with sim($n$) to get act$_C$.
20:    Define act$_L = (act.a, act_L.I)$ and act$_C = (act.a, act_C.I)$;
21: return $γ, τ$

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 all at once thereby ensuring that a quorum agreement on a state component in the larger system is maintained in the cutoff system.²

VI. EXPERIMENTAL RESULTS

We have applied the proposed strategy on a variety of different distributed protocols³,⁴ 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 $\varphi_{init}(γ_L), \varphi_{step}(γ_L, τ_L)$ and $\varphi_{safety}(γ_L)$. We reduce the problem of checking correctness to satisfiability of the generated FOL formulae. For example, for checking the $\varphi_{step}(γ_L, τ_L)$ condition which is a condition of the type $p \implies q$ to be correct, we check whether $p \land \neg q$ is unsatisfiable. We use Z3 [14] 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.

VII. RELATED WORK AND CONCLUSION

In the recent past, there has been a lot of interest in automated and mechanised verification of distributed protocols.

²We provide an example of our technique for consensus protocols on the Toy Consensus protocol in the full version of our paper at https://github.com/shreesha00/FMCAD.git
³The RML descriptions and the SMT encoding of the simulation relation and cutoff protocol for each protocol can be found at the following link: https://github.com/shreesha00/FMCAD.git
⁴We provide detailed descriptions of each protocol and its cutoff instance in the full version of our paper at https://github.com/shreesha00/FMCAD.git
We note that our approach has limitations. In particular, 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 \( \varphi \) 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


| Protocol                     | Cutoff | Time Taken(s) | \( |\gamma|\) |
|------------------------------|--------|---------------|---------|
| Sharded Key-Value Store[15]  | 2      | 0.02          | 5       |
| Leader Election in a Ring[16]| 2      | 0.03          | 4       |
| Centralized Lock Server[17]  | 2      | 0.02          | 5       |
| Lock Server Sync[18]         | 2      | 0.01          | 2       |
| Ricart Agrawala[19]          | 2      | 0.01          | 6       |
| Two Phase Commit[20]         | 2      | 0.02          | 9       |
| Toy Consensus ForAll[18]     | 1      | 0.07          | 5       |
| Consensus[18]                | 2      | 29.7          | 11      |

TABLE II: \( \gamma \) 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.


