Counterexample Driven Quantifier Instantiations with Applications to Distributed Protocols

ORR TAMIR, Tel Aviv University, Israel
MARCELO TAUBE, Tel Aviv University, Israel
KENNETH L. MCMILLAN, University of Texas at Austin, USA
SHARON SHOHAM, Tel Aviv University, Israel
JON HOWELL, VMware Research, USA
GUY GUETA, VMware Research, Israel
MOOLY SAGIV, Tel Aviv University, Israel

Formally verifying infinite-state systems can be a daunting task, especially when it comes to reasoning about quantifiers. In particular, quantifier alternations in conjunction with function symbols can create function cycles that result in infinitely many ground terms, making it difficult for solvers to instantiate quantifiers and causing them to diverge. This can leave users with no useful information on how to proceed.

To address this issue, we propose an interactive verification methodology that uses a relational abstraction technique to mitigate solver divergence in the presence of quantifiers. This technique abstracts functions in the verification conditions (VCs) as one-to-one relations, which avoids the creation of function cycles and the resulting proliferation of ground terms.

Relational abstraction is sound and guarantees correctness if the solver cannot find counter-models. However, it may also lead to false counterexamples, which can be addressed by refining the abstraction and requiring the existence of corresponding elements. In the domain of distributed protocols, we can refine the abstraction by diagnosing counterexamples and manually instantiating elements in the range of the original function. If the verification conditions are correct, there always exist finitely many refinement steps that eliminate all spurious counter-models, making the approach complete.

We applied this approach in Ivy to verify the safety properties of consensus protocols and found that: (1) most verification goals can be automatically verified using relational abstraction, while SMT solvers often diverge when given the original VC, (2) only a few manual instantiations were needed, and the counterexamples provided valuable guidance for the user compared to timeouts produced by the traditional approach, and (3) the technique can be used to derive efficient low-level implementations of tricky algorithms.

CCS Concepts: • **Theory of computation → Logic and verification**; **Automated reasoning**; **Abstraction**.

Additional Key Words and Phrases: Formal verification, Ivy, SMT, Abstraction-refinement

ACM Reference Format:
Orr Tamir, Marcelo Taube, Kenneth L. McMillan, Sharon Shoham, Jon Howell, Guy Gueta, and Mooly Sagiv. 2023. Counterexample Driven Quantifier Instantiations with Applications to Distributed Protocols. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 288 (October 2023), 27 pages. https://doi.org/10.1145/3622864

Authors' addresses: Orr Tamir, ortamir@post.tau.ac.il, Tel Aviv University, Israel; Marcelo Taube, mail.marcelo.taube@gmail.com, Tel Aviv University, Israel; Kenneth L. McMillan, kenmcmil@gmail.com, University of Texas at Austin, USA; Sharon Shoham, sharon.shoham@gmail.com, Tel Aviv University, Israel; Jon Howell, howell@vmware.com, VMware Research, USA; Guy Gueta, ggolangueta@vmware.com, VMware Research, Israel; Mooly Sagiv, msagiv@post.tau.ac.il, Tel Aviv University, Israel.

2475-1421/2023/10-ART288
https://doi.org/10.1145/3622864

1 INTRODUCTION

Distributed systems are notoriously hard to get right. Subtle bugs appear from design to implementation. Thus formally verified software implementation is of growing interest to the systems community. Ideally, one would like to have fully automated verification. To date, however, automated verification of realistic implementations of distributed protocols has proved elusive. As a practical compromise, it is common to make a division of labor between human and machine: the human proposes an inductive invariant, and the machine checks the correctness of this invariant. This reduces the machine's task to check the satisfiability of a first-order formula (modulo some elementary theories). This more circumscribed problem can be handled by SMT solvers, which are by now well-developed and efficient.

All is not entirely well, however. Modern SMT solvers struggle with quantifiers over unbounded sets. Such quantifiers are very much needed for reasoning about distributed systems since the system consists of an unbounded set of processes, as well as unbounded data structures. Solvers handle quantifiers by heuristically instantiating variables with ground terms. In practice, this frequently leads to solver timeouts that are extraordinarily difficult to debug and correct; the solver does not provide actionable feedback to the user [Hawblitzel et al. 2017, Section 6.3.2]. In response to a timeout, the user must add trigger patterns to guide the solver in instantiating quantifiers, or else introduce abstractions that make the problem easier for the solver. With no useful feedback and little understanding of the internals of the solver, this is a hit-or-miss process at best. Furthermore, because the prover relies on fragile heuristics, small changes in the lemmas that inevitably occur in the development of the proof can cause proofs that formerly succeeded to fail unpredictably [Hawblitzel et al. 2017; Leino and Pit-Claudel 2016]. Instability is the most frustrating aspect of interacting with a solver.

To remedy this situation, we propose an approach that shifts the division of labor between human and machine in a way that allows the machine to produce actionable feedback without unduly increasing the effort required of the human. This approach is based on the observation that a common cause of solver timeouts is *function cycles* occurring in the verification condition. These are formed from function symbols introduced by the user, as well as Skolem functions introduced by the prover to eliminate quantifier alternations. Function cycles can cause heuristic quantifier instantiation to diverge, whereas, in their absence, the SMT solver is a decision procedure and behaves more stably. Moreover, without cycles, the solver is able to produce true finite counterexamples as feedback.

Previous work [Padon et al. 2017a, 2016; Taube et al. 2018] has already recognized the benefits of avoiding function cycles in the verification conditions. There, the user is required to specify the protocol and its invariants such that the verification condition formulas are *stratified*— contains no function cycles. To do so, the user may employ rewrites and derived relations [Padon et al. 2017a], or leverage modular reasoning [Taube et al. 2018] to break the verification conditions into stratified ones. Unfortunately, finding the right rewrites or partition into modules may be extremely difficult, and is left entirely to the user, without any guidance from the tool. In fact, it is not clear if rewrites or modularity always suffice (for example, we did not succeed to apply these methodologies to verify PBFT [Castro and Liskov 2002]). The key challenge is that the fragment of stratified formulas is not compositional—a Boolean combination of stratified formulas is not necessarily stratified.

We propose an alternative approach that does not require the user to manually eliminate function cycles by changing the specification of the protocols, nor to provide trigger patterns in response to timeouts in the presence of function cycles. First, we automatically abstract the verification condition in a way that eliminates function cycles but may result in spurious counterexamples. The user analyzes these counterexamples in order to refine the abstraction. Thus, instead of timing out, the solver provides actionable feedback. We argue that this approach provides a more effective form

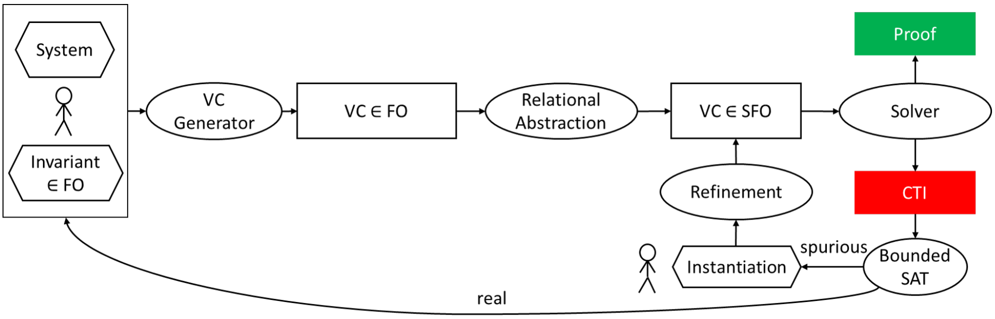


Fig. 1. Verifying infinite-state systems with user-guided quantifier instantiation. Bounded SAT is a standard enumeration procedure for checking models of bounded size. FO denotes a first-order formula. SFO dentes a stratified first-order formula, without functions cycles, guaranteeing the convergence of SMT solvers.

of interaction between the user and the tool, reducing human effort and resulting in more stable proofs. Our abstraction eliminates function cycles by replacing function symbols with relation symbols. The user refines this abstraction by instantiating the totality axioms for these relations, yielding a complete proof approach. While the idea of abstracting functions as relations is not new, we are not aware of other works that used it in conjunction with user-provided instantiations of totality axioms in a systematic way as we do in this work.

To test this approach, we apply it to verify implementations of PBFT [Castro and Liskov 2002]. PBFT stands for practical Byzantine Fault tolerance consensus protocol. We use the extraction mechanism of [McMillan 2020] to generate executable code from the verified model. Extracted code for the basic implementation has performance comparable to existing implementations, delivering throughput within 66% of Castro and Liskov's PBFT implementation under metropolitan-area network delays. Few manual instantiations were needed in these applications and the counterexamples provided valuable guidance for the user. The proofs achieve an annotation: code ratio of 1.3:1, a lower burden than in prior proofs of similar systems.

Main Contributions. The paper makes the following contributions: (i) A systems engineering methodology for predictably exploiting high automation in deductive verification, enabling developers to operate in a predictable, diagnosable framework that requires neither writing automation tactics nor diagnosing unpredictable heuristic failures. (ii) Verified variants of PBFT, as well as automatically extracted implementations thereof, demonstrating that the methodology scales to 700 lines of code. (iii) An evaluation showing that the methodology emits reasonably efficient executable implementations. (iv) Crisp proofs of Paxos and Synchronized BFT consensus protocols.

2 OVERVIEW

In this section, we demonstrate our methodology on a toy protocol for consensus, taken from [Taube et al. 2018]. The toy consensus protocol is naive and does not provide any liveness guarantees. We use it for illustration purposes only as it nicely demonstrates all the ingredients of our approach. Furthermore, despite its simplicity, it uses the concept of set majorities, which is a key idea in many realistic protocols, including Raft [Ongaro and Ousterhout 2014] and Multi-Paxos [Lamport 2002a].

A full example of an instantiation step used during the verification of PBFT is described in Section 5.

```
alreadyvoted := false
voters[] := [∅,...,∅]

upon client_request() do
 if ¬alreadyvoted
  send request_vote_msg(self)
  alreadyvoted := true
  voters[myvalue] := {myid}

upon recv(msg) do
 if msg.type = request_vote_msg ∧
         ¬alreadyvoted
  alreadyvoted := true;
  send vote_msg(myvalue, msg.src)
 elif msg.type = vote_msg
   voters[msg.value].add(msg.src)
 if |voters[msg.value]| > N/2
  send decided_msg(msg.value)
```

Fig. 2. A toy consensus protocol.

Relation \ Function	**Description**
none : value	Null value
vote(N : node) : value	Node vote
decision(V : value)	Decided value
member(N : node, Q : quorum)	$N \in Q$

Majority Intersection Axiom:$\forall Q_1, Q_2 :$ quorum. $\exists N :$ node.member$(N, Q_1) \wedge$ member(N, Q_2)

Fig. 3. FOL modeling of the toy consensus protocol.

```
init
assume ∀N: node. vote(N) = none;
assume ∀V: value. decision(V) = false;

action cast_vote(n: node, v: value)
 assume vote(n) = none ∧ v ≠ none;
 vote(n) := v;

action decide(v: value)
 assume v ≠ none ∧ ∃Q as Q̂: quorum. ∀N: node. member(N,Q) → vote(N) = v;
 decision(v) := true;
```

Fig. 4. Actions for toy consensus.

2.1 Toy Consensus Protocol

Figure 2 shows the pseudo code of the toy consensus protocol. Every node that participates in the protocol executes this code, where the number of nodes is a parameter of the protocol. The protocol allows each node to cast a vote to a value. When a value receives a majority of the votes, it is decided. The protocol is designed to ensure that at most one value is decided. This is a *safety* property, whose verification we will use to demonstrate our methodology.

2.2 First Order Modeling

In order to verify the desired property, the user expresses the protocol in a modeling language such as the one provided by Dafny [Leino 2017], Alloy [Jackson 2019] and Ivy [McMillan 2020], which is based on first order logic. The user also writes the desired property in first order logic and provides additional candidate invariants as the starting point for interactively constructing a verified inductive invariant.

When modeling the protocol in pure first-order logic[1], the state space is captured as a set of sorted first-order constants, relations, and functions. For the toy example, these are listed in Figure 3. The node and value sorts (types) represent nodes and values, respectively. The vote function indicates the vote of each node; initially, all votes are for the value none, indicating that no node has voted. The decision relation represents the set of values that have been decided; initially, this is the empty set. Set majorities are modeled by the abstract notion of *quorums*, captured by the quorum sort, where the member relation indicates for each node n and quorum q whether n belongs to q (this is a many-to-many relation). The property of majorities that the correctness of the protocol relies on is captured by the Majority Intersection axiom, which ensures that any pair of quorums have a non-empty intersection.

The behavior of the protocol is defined by actions that update the relations and functions. In the toy consensus, this is captured by the actions cast_vote and decide, see Figure 4. The former updates the vote of a node to a value provided that it has not yet voted. The latter makes a decision on a value if a quorum of nodes all voted for the value. Actions are executed non-deterministically, in an interleaving manner, where each action is atomic.

The desired safety property, is that the set of decided values never includes more than one value, is also specified as a first-order formula (see Figure 6). This property is indeed an *invariant* of the protocol—it holds for all the reachable states—but it is not preserved by the actions of the protocol, i.e., it is not *inductive*. A formula is an *inductive invariant* if (i) **(Initiation)** it holds initially, and (ii) **(Consecution)** it is preserved by the transitions of the system. Figure 6 presents two additional invariants, which together with the safety property comprise an *inductive invariant*, thus establishing the safety of the protocol.

2.3 Verification Conditions

From the system description and the invariants, *verification conditions* (VCs) in first-order logic are automatically extracted, such that unsatisfiability of the VC formulas indicates that the property is verified. In the simple case of a single module[2] as in the toy example, the VC formulas encode violation of the initiation or consecution requirements. Specifically, the following VC formulas are generated for each individual invariant I and for each action act:

- *Violation of initiation:* $Axioms \wedge Init \wedge \neg I$, where $Axioms$ denotes the conjunction of all axioms that are used to define the set of states and $Init$ denotes the initial condition.
- *Violation of consecution:* $Axioms \wedge Inv \wedge Tr_{\mathrm{act}} \wedge \neg I'$, where Inv is the conjunction of *all* invariants assumed in the pre-state, Tr_{act} is the transition formula associated with act, relating pre-states defined over $\mathcal{V}$ with post-states defined over $\mathcal{V}' = \{v' \mid v' \in \mathcal{V}\}$, and I' is obtained from I by substituting each $v \in \mathcal{V}$ by $v' \in \mathcal{V}'$, representing the invariant in the post-state.

[1] Our technique can also be applied to handle theories, e.g., by using the MBQI method [Ge and de Moura 2009].

[2] We use a toy "monolithic" example for simplicity of the exposition. Our approach, however, is not restricted to a specific construction of the verification conditions. For example, in our evaluation, we also consider modular specifications of protocols, where the verification conditions are constructed as in [Taube et al. 2018].

Note that an invariant I appears in a consecution VC formula both positively (as part of Inv in the pre-state, defined over $\mathcal{V}$) and negatively as $\neg I'$ (in the post-state, defined over $\mathcal{V}'$), restricting the use of $\forall^*\exists^*$ and $\exists^*\forall^*$ formulas. We will discuss this in the next subsection.

Inv is an inductive invariant if the VC formulas are unsatisfiable for every I and act, in which case the system is verified. Otherwise, a model of the satisfiable VC formula comprises a counterexample to induction (CTI). For example, the consecution VC formula for the Safety invariant of the toy example w.r.t. decide is satisfiable when only the safety is assumed in the pre-state. This means that Safety is not inductive by itself, and needs to be strengthened in order to be inductive. A CTI is displayed in Figure 5. The CTI depicts an unreachable state: v_0 was decided but no node voted for it, and it is indeed eliminated by the additional invariants from Figure 6, which together comprise an inductive invariant for the toy consensus protocol. In all figures, objects of different shapes denote the elements in the domains of different sorts; edges denote the interpretation of unary functions and binary relations; boxes depict unary relations; constants are displayed above the elements assigned to them.

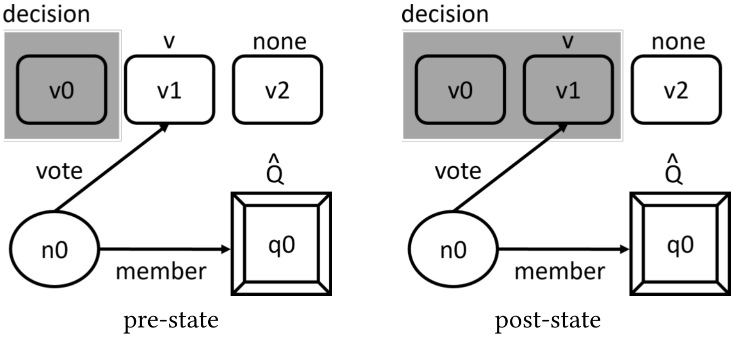


Fig. 5. A CTI for consecution of the Safety invariant w.r.t. the decide action.

2.4 Challenge: Sort Cycles

The use of first-order logic over uninterpreted sorts abstracts certain aspects of the system such as integers and floating-point numbers. It enables the generation of verification conditions over uninterpreted first-order logic. However, discharging VCs in *arbitrary* first-order logic is still extremely challenging due to functions and quantifier alternations that lead to sort cycles. Specifically, a formula $\forall x : s_1.\ \exists y : s_2.\ \psi(x, y)$, respectively, a function $f : s_1 \rightarrow s_2$, create a dependency edge from sort s_1 to s_2, indicating that for every term of sort s_1, we may need to consider a unique term of sort s_2. These dependencies may result in cycles. Sort cycles make the space of ground terms infinite. As a result, finding the right quantifier instantiations to discharge the VCs is tremendously difficult (and undecidable in general).

In practice, natural modeling often leads to sort cycles. For example, in the toy consensus protocol, the consecution VC of the Safety invariant w.r.t. the decide action includes a sort cycle, depicted in Figure 7. In this example, the majority intersection property ensures that *for every* two quorums there *exists* a common node. In addition, the function vote ensures that *for every* node there *exists* a value for which it voted. Finally, the QuorumSupport invariant states that *for every* decided value v, there *exists* a quorum in which all nodes voted to v. The verification condition that encodes consecution of the Safety invariant includes all these assumptions, together with the negation of the invariant in the post-state. Unfortunately, the cycle between the sorts quorum-node-value-quorum means that there are infinitely many possible instantiations.

Invariant Name	**Formula**
Safety	$\forall V_1, V_2 : \text{value}.\ \text{decision}(V_1) \land \text{decision}(V_2) \rightarrow V_1 = V_2$
NullValue	$\neg\text{decision}(\text{none})$
QuorumSupport	$\forall V : \text{value}.\ \exists Q : \text{quorum}.\ \forall N : \text{node}.\ \text{decision}(V) \land \text{member}(N, Q) \rightarrow \text{vote}(N) = V$

Fig. 6. Invariants for the toy consensus protocol.

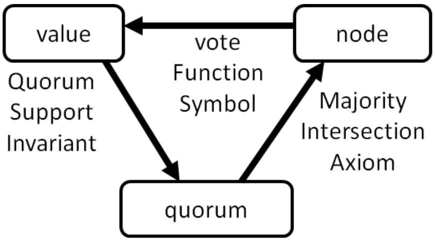


Fig. 7. The sort graph induced by the VC of the Safety invariant and the decide action in the toy example.

On the other hand, if the verification conditions contain no sort cycles, they belong to the fragment of *stratified formulas* [Ge and de Moura 2009], in which case, there are only finitely many instantiations, and discharging the verification conditions (i.e., checking that the formulas are unsatisfiable) is decidable. Indeed, SMT solvers such as Z3 [de Moura and Bjørner 2008] behave as a decision procedure for these formulas. (Our approach is not limited to specific solver heuristics, as long as the solver acts as a decision procedure for the decidable fragment of stratified formulas.) Furthermore, when a VC in this fragment is violated (i.e., the VC formula is satisfiable), it always has a *finite* model that the solver can display to the user as a counterexample.

This is the case when only the Safety invariant is considered in the toy example, without the supporting invariants—the invariant is purely universally quantified, hence it does not contribute any dependencies between sorts. Unfortunately, the Safety invariant is not inductive on its own, hence a CTI exists in this case (as mentioned earlier), and additional invariants are required.

2.5 Our Approach

Our methodology aims at providing a user with guidance, stability, and diagnosability in *every step* of the verification process. Stability and diagnosability are achieved similarly to [Padon et al. 2017a, 2016; Taube et al. 2018] by ensuring that all of the satisfiability queries made to an automatic solver belong to the fragment of stratified formulas, which is theoretically decidable, enjoys support from mature solvers, and enables accompanying each failed check with a small finite counterexample. The key novelty of our methodology is that, as opposed to previous work, we do not require the user to rewrite or decompose the model until the VCs fit into the decidable fragment. Instead, we perform a *relational abstraction* that achieves this goal automatically. The relational abstraction abstracts a user-selected subset of the functions, whose removal makes the sort graph acyclic, into (functional) relations. When the abstraction is imprecise, spurious counterexamples to induction are obtained, in which case *refinement* is employed based on user-provided instantiations. In order to distinguish spurious CTIs from real ones, we perform *bounded* satisfiability checks of the concrete VC whose abstraction produced the CTI. In this way, the user receives constructive feedback at every step of the verification process. The full process of formal verification using our methodology is displayed in Figure 1. Importantly, we show that the approach is complete: if the VC formulas are unsatisfiable, there always exists a finite set of refinements (instantiations) that make the abstraction sufficiently precise to discharge the VCs.

2.6 Eliminating Sort Cycles by Relational Abstraction

Cycles in the sort-graph are introduced by functions and by quantifier alternations. In fact, the latter may also be understood as functions by means of *Skolemization*: $\forall x : s_1.\ \exists y : s_2.\ \psi(x, y)$ is equi-satisfiable to $\forall x.\ \psi(x, f(x))$, where f is a fresh function symbol, called a *Skolem function*. Relations (and constants), on the other hand, do not create dependencies between sorts. The key idea to eliminating cycles is therefore to specify functions (including Skolem functions) as relations. Indeed, a function is a special case of a relation that satisfies two additional properties:

Functionality: an element in the domain is mapped to *at most* one element in the range.
Totality: for every element in the domain there *exists* an image.

Thus, instead of using atoms such as $f(x) = y$ in the VCs, for $f : s_1 \rightarrow s_2$, one can use $r_f(x, y)$, where $r_f : s_1 \times s_2$ is a fresh relation symbol axiomatized to be functional and total. Modeling functions as relations together with the two properties above does not incur any abstraction. Specifically, applying such a transformation on the VC formulas preserves (un)satisfiability. However, if one (or both) of the properties is omitted, the protocol's VCs may be spuriously violated.

For example, consider the function $\text{vote}(N : \text{node}) : \text{value}$ that introduces the edge from node to value in the sort graph from Figure 7. If we specify it as a relation $r_{\text{vote}}(N : \text{node}, V : \text{value})$ and do not assume the functionality property, the corresponding VC formulas may be violated by a pre-state where a node voted for multiple values, allowing for two values to be decided (even though such a state is not reachable). However, if we assume the functionality property, correctness is restored, even without assuming the totality property. That is, in the case of vote, a relational abstraction that captures the functionality property but not the totality property is sufficiently precise to discharge the VCs.

2.7 Relational Abstraction

Fortunately, the functionality property is purely universally quantified, and hence adding it as an axiom does not introduce any dependencies between sorts. In contrast, the totality property introduces $\forall\exists$ quantifier alternation that captures the dependency of the image's sort on the domain's sort. To avoid the latter, the *relational abstraction* automatically abstracts a function into a relation that is axiomatized to be functional, but not necessarily total. In other words, instead of *total* functions, the abstraction considers *partial* functions. This eliminates sort cycles and ensures that the abstract VC formulas are stratified.

Importantly, the abstraction is carried out *automatically* on the VC formulas, which are also automatically generated. Note that when abstracting a function f into a relation r_f, we do not *verify* that r_f is functional. Instead, we use functionality as an axiom, since we are only interested in discharging the VCs for the case where r_f is functional (as it is in the concrete model).

The relational abstraction is sound: the set of all interpretations of a functional relation is an overapproximation of the set of all interpretations of a function. Hence, if correctness (which in our context means unsatisfiability of the VC formulas) holds for the former, it is guaranteed to hold for the latter.

As demonstrated by the abstraction of the vote function, the correctness of the concrete model often carries over to the abstraction. Indeed, in the toy example abstracting the vote function suffices to eliminate the sorts cycle, while being precise enough to establish correctness.

2.8 Imprecision of the Abstraction

The relational abstraction is, however, bound to be imprecise in some cases. As an example, consider again the toy consensus protocol, but this time consider applying the relational abstraction on the Skolem function that is introduced for the ∀∃ quantification in the majority intersection axiom. Skolemization happens automatically and is hidden from the user, but for the sake of illustration, we present it here explicitly.

Namely, Skolemization introduces a function $\mathit{inter}(Q_1 : \text{quorum}, Q_2 : \text{quorum}) : \text{node}$, and rewrites the axiom into:

$$\forall Q_1, Q_2 \colon \text{quorum. member}(\mathit{inter}(Q_1, Q_2), Q_1) \wedge \text{member}(\mathit{inter}(Q_1, Q_2), Q_2) \tag{1}$$

The function *inter* is responsible for the edge from quorum to node in the sort graph. Just like vote, abstracting it is also sufficient for eliminating the sorts cycle in this example. The relational abstraction (automatically) transforms *inter* into a relation $r_{inter}(Q_1 : \text{quorum}, Q_2 : \text{quorum}, N : \text{node})$, thus further rewriting the majority intersection property into

$$\forall Q_1, Q_2 \colon \text{quorum. } \forall N : \text{node. } r_{inter}(Q_1, Q_2, N) \rightarrow \text{ member}(N, Q_1) \wedge \text{ member}(N, Q_2) \tag{2}$$

where r_{inter} is axiomatized to be functional (but not necessarily total). Comparing the transformed axiom to the original one reveals that, essentially, the relational abstraction has replaced the existential quantification over N in the majority intersection property by a guarded universal quantification.

Unfortunately, when r_{inter} is not assumed to be total, the VC for the consecution of the Safety invariant w.r.t. the decide action in Fig. 4 cannot be discharged, even when assuming all invariants from Figure 6 in the pre-state, since the abstract VC formula is satisfiable (even though the actual VC formula is unsatisfiable). The counterexample to induction in Fig. 8 is obtained.

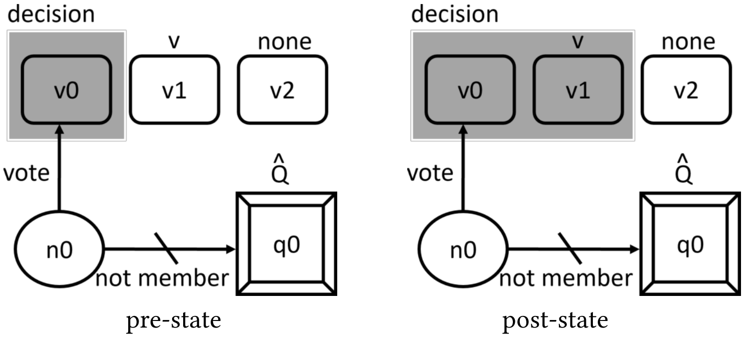
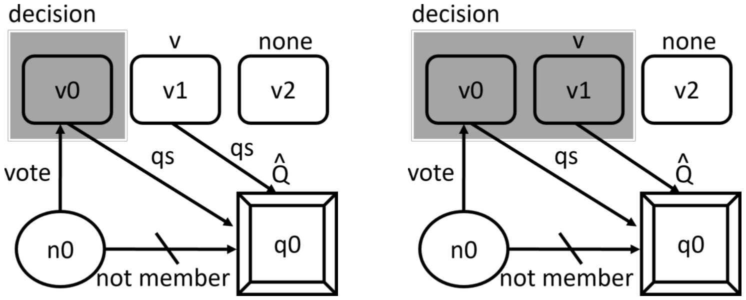


Fig. 8. A (spurious) CTI for the Safety invariant w.r.t. the decide action, when the pre-state assumes all the invariants from Figure 6 but considers the abstracted majority intersection axiom. decision(v_1) is changed from false to true by the action, thus violating the safety invariant in the post-state.

The counterexample in Fig. 8 depicts a pre-state where a value v_0 has already been decided and has a supporting quorum q_0 (which supports v_0 vacuously as it has no nodes in it, as seen by the interpretation of the member relation), but the decide action is called on a distinct value v_1 (v = v_1) that also has q_0 as a supporting quorum (which, again, supports it vacuously as it has no nodes in it). The relation r_{inter} is empty, thus not enforcing that a quorum must not be empty (which is implied by the requirement that every two quorums intersect). As a result, in the post-state the values v_0, v_1 are both decided, violating Safety. Clearly, the CTI is spurious: it does not indicate a



pre-state post-state

Fig. 9. An extension of the CTI from Figure 8, in which the Skolem function for QuorumSupport, denoted qs for short, is displayed explicitly.

violation of the concrete VC, which is in fact unsatisfiable (recall that the invariants in Figure 6 are inductive).

2.9 Refinement via Instantiations

Having examined the counterexample to induction displayed in Figure 8, the user realizes that the pre-state violates the majority intersection axiom. This is indeed an outcome of the relational abstraction, which does not require r_{inter} to be total. To overcome it, the user may *refine* the abstraction by introducing *instantiations* of the totality axiom to be used when verifying the VC generated for Safety w.r.t. decide. While the totality axiom introduces sort dependencies that in this example create a cycle in the sort-graph, its instantiations are existentially quantified and do not add sort dependencies.

Instantiations are best explained by an example. For convenience of presentation, we shall assume that when a ∀∃ invariant is Skolemized, the Skolem function produced is named after the name of the invariant. For example, the QuorumSupport invariant shown in Fig. 6 introduces the Skolem function $qs(V : \text{value}) : \text{quorum}$ and is rewritten to

$$\forall V\colon \text{value}.\ \forall N\colon \text{node}.\ \text{decision}(V) \land \text{member}(N, qs(V)) \rightarrow \text{vote}(N) = V$$

(Note that qs, exactly like the ∀∃ quantification that introduced it, does not create sort cycles and hence was not abstracted.) Accordingly, Figure 9 extends the CTI from Figure 8 to also interpret qs. Figure 10 presents a user-provided instantiation block that refines the relational abstraction of the majority intersection property for the sake of eliminating the CTI and discharging the abstract VC generated for the Safety invariant w.r.t. the decide action. The instantiate block provides two instantiations of the totality axiom associated with r_{inter} on $\widehat{Q}, qs(V_1)$ and on $\widehat{Q}, qs(V_2)$, where $\widehat{Q}$ is defined in the decide action and V_1, V_2 denote the variables used to specify the negation of the Safety invariant in the post-state—these variables are existentially quantified in the negation of the Safety invariant, and are hence replaced by Skolem constants.

To come up with these instantiations, the user examines the CTI in Figure 9 and realizes that the (abstract) pre-state is not feasible since the quorum supporting $V_1 = v_0$, which is already decided, is (V_1), and the quorum supporting $V_2 = v_1$, which is the argument of decide and is currently being decided, is $\widehat{Q}$, and these two quorums are the same, despite v_0 and v_1 being distinct, which is only possible since the quorums are empty, and, in particular, have an empty intersection. Thus, to eliminate the CTI, the totality axiom of r_{inter} is instantiated on $\widehat{Q}, qs(V_1)$. This precisely mimics

```
invariant[Safety]
$\forall V_1, V_2$: value. decision($V_1$) $\land$ decision($V_2$) $\rightarrow$ $V_1 = V_2$
instantiate for action decide
total[inter]($\widehat{Q}$,qs($V_1$))
total[inter]($\widehat{Q}$,qs($V_2$))
```

Fig. 10. Instantiations of totality of r_{inter} for the Safety invariant and decide action.

and excludes the scenario depicted in the counterexample. The instantiation of r_{inter} on $\widehat{Q}$, (V_2) is needed to eliminate the symmetrical CTI where the roles of V_1 and V_2 are swapped.

The instantiation block refines the VC formula for decide by adding to it the following constraint: $(\exists N_1 : \text{node}.\ r_{inter}(\widehat{Q}, qs(V_1), N_1)) \land (\exists N_2 : \text{node}.\ r_{inter}(\widehat{Q}, qs(V_2), N_2))$.

With the addition of these constraints, the (abstract) VC becomes unsatisfiable. Namely, the axiom in Equation (2) ensures that the instantiations imply that $\text{member}(N_1, qs(V_1)) \land \text{member}(N_1, \widehat{Q})$ and $\text{member}(N_2, qs(V_2)) \land \text{member}(N_2, \widehat{Q})$ both hold, which in turn implies that either $\text{vote}(N_1) = V_1$ and $\text{vote}(N_1) = V_2$ (if V_2 is the argument of decide) or $\text{vote}(N_2) = V_1$ and $\text{vote}(N_2) = V_2$ (if V_1 is the argument of decide), in contradiction to the property that $V_1 \neq V_2$, which is asserted by the negation of Safety. Thus, verification is complete. This reasoning, establishing that the refined VC is unsatisfiable, is performed completely automatically by the SMT solver, without the involvement of the user. The user need not understand the low level details of the proof. All the user needed to do was provide the instantiations. Note that we simplified our syntax for clarity while we allow nested instantiations when needed. Notably, using instantiations allows us to overcome abstracted function symbols. We discuss this formally in Section 4.

2.10 Distinguishing Spurious CTIs From Real Ones

The ability to display CTIs in every step is extremely useful for helping the user make progress in the verification process. Still, diagnosing CTIs is a challenging task. This is largely because CTIs can have several sources, each of which needs to be addressed differently: (i) they may be real CTIs, indicating that the invariants do not form an inductive invariant, thus the invariants (or the protocol) need to be modified (e.g., by adding an invariant), or (ii) they may be spurious CTIs, indicating that the relational abstraction needs to be refined by adding instantiations that will allow to discharge the (abstract) VCs. The main difficulty is that, given a CTI, the user does not a-priori know which is the case.

In order to allow the user to focus their efforts in the right direction, we employ a *bounded* (un)satisfiability check of the failed VC to distinguish between the two aforementioned cases. Namely, we consider the *concrete* VC formula whose abstraction resulted in the CTI and we check whether it is satisfiable over fixed finite domains, as done by TLA [Lamport 2002c] and Alloy [Jackson 2019]. To do so, we add axioms that restrict the domain (for example, stating that there are at most 3 nodes and 2 values). From our experience, real counterexamples to induction are small and thus found by bounded satisfiability checks with small bounds.

3 PRELIMINARIES

In this section we provide a brief background on first-order logic and the decidable fragment used in this work. We consider many-sorted first-order logic with equality, where formulas are defined over a many-sorted vocabulary.

Vocabulary. Let $\mathcal{V} = \langle S, C, F, R\rangle$ be a vocabulary where: S is a finite set of sorts, C is a finite set of constant symbols, F is a finite set of function symbols $f\colon s_1 \times \cdots \times s_n \rightarrow s_{n+1}$, R is a finite set of relation symbols $r : s_1 \times \cdots \times s_n$.

From here on, $\mathcal{V}$ is fixed unless otherwise stated.

Terms and Formulas. Terms t and formulas φ over $\mathcal{V}$ are defined in the usual way:

$$\begin{aligned} t &::= x \mid c \mid f(t_1, \dots, t_n) \\ \varphi &::= a \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \forall x : s.\ \varphi \mid \exists x : s.\ \varphi \\ a &::= r(t_1, \dots, t_n) \mid t_1 = t_2 \end{aligned}$$

where x is a logical variable, $c \in C$, $f \in F$ and $r \in R$. a denotes an *atomic formula*, which is either an application of a relation symbol or an equality. Sorts extend to terms in the usual way, and function and relation applications are required to obey the sort constraints. We omit the sorts from the notation when they are irrelevant or clear from the context. $\varphi \rightarrow \psi$ is a shorthand for $\neg\varphi \vee \psi$. A *literal* is an atomic formula or its negation. A term, respectively a formula, is *ground* if it includes no variables. A formula is *closed* if it includes no free variables. Structures, as well as the satisfiability and validity of formulas, are defined in the usual way. For closed formulas φ, ψ we write $\varphi \Longrightarrow \psi$ if the formula $\varphi \rightarrow \psi$ is valid.

Normal Forms. A formula is in *negation normal form* (NNF) if negation is restricted to atomic formulas (where $\alpha \rightarrow \beta$ is understood as $\neg\alpha \vee \beta$). It is *universally quantified* (respectively, *existentially-quantified*) if it is in NNF and includes only universal (respectively, existential) quantifiers. The *universal* (respectively, *existential*) *closure* of a formula is the formula obtained by adding a prefix of universal (respectively, existential) quantifiers for all the free variables in the formula.

Skolemization. For a formula φ in NNF, we consider its *skolemization* $SK(\varphi)$, which is a universally quantified formula over vocabulary $\mathcal{V}' = \langle S, C', F', R\rangle$ with extra constant and function symbols. Skolemization preserves satisfiabilty: φ is equi-satisfiable to $SK(\varphi)$.

Stratified Formulas. This work uses the fragment of *stratified formulas*—a fragment of many-sorted first-order logic for which checking satisfiability is decidable, and Z3 is a decision procedure. We briefly describe this fragment. More details can be found in [Ge and de Moura 2009; Taube et al. 2018]. A formula φ in NNF is *stratified* if its *sort graph* has no cycles containing the sort of some universally-quantified variable. The sort graph of φ is a graph in which the vertices are sorts and we draw an arc from sort s to sort t if there is a function symbol $f : s_1 \times \ldots \times s \times \ldots \times s_n \rightarrow t$ in $SK(\varphi)$. (Note that an existential quantifier of sort t in the scope of a universal quantifier of sort s in φ will also introduce an arc in the sort graph of φ due to the Skolem function used in $SK(\varphi)$.)

The key to the decidability of stratified formulas is that, by Herbrand's theorem, a universally quantified formula is equi-satisfiable to its set of *ground instances*, obtained by instantiating universally quantified variables by ground terms (for formulas that include equality, this holds provided that the equality axioms are added to the formula). In the case of stratified formulas, the acyclicity of the sort graph ensures that the set of ground terms, and hence of ground instances, of the skolemized formula, is finite. This also ensures that when a stratified formula is satisfiable, it always has a finite model.

Verification Conditions in First Order Logic. Given a vocabulary $\mathcal{V}$, a state is a first-order structure over $\mathcal{V}$, possibly restricted by a set of *axioms*—closed first-order formulas over $\mathcal{V}$. The initial states are given by a closed formula over $\mathcal{V}$, and the steps (transitions) of the system are expressed by closed first-order formulas that capture how the state is updated. Invariants are also specified by closed first-order formulas. From these, *verification condition* (VC) formulas in first-order logic are constructed such that the unsatisfiability of the VC formulas ensures the correctness of the invariants [Taube et al. 2018]. The VCs are discharged by establishing that the VC formulas are indeed unsatisfiable. (When the VC formulas are unsatisfiable, it is often said that the VCs are *valid.* We avoid using this terminology in this paper to prevent confusion.)

4 RELATIONAL ABSTRACTION AND REFINEMENT

We present the relational abstraction whose goal is to transform the VC formulas into stratified formulas, for which checking (un)satisfiability is decidable. The abstraction is sound but incomplete. That is, if the abstract formula is unsatisfiable, so is the original formula, but the converse does not necessarily hold. Completeness may be recovered by introducing finitely many *instantiations* of totality axioms that refine the abstraction.

We fix a vocabulary $\mathcal{V}$ and a formula φ over $\mathcal{V}$ whose unsatisfiability we wish to prove. We assume that φ is in a prenex universal form (only to simplify the presentation), that is, it is closed and consists of a prefix of universal quantifiers, followed by a quantifier-free formula. This is without loss of generality since any formula can be converted to an equi-satisfiable formula of this form by Skolemization. We further fix a set $F \subseteq \mathcal{V}$ of non-nullary function symbols from $\mathcal{V}$. In practice, φ would be the skolemized verification condition formula, and F would be a set of function symbols whose removal makes the sort graph of φ acyclic.

In the rest of the section we define the relational abstraction (Section 4.1) and its refinement via instantiations of totality axioms (Section 4.2).

4.1 Relational Abstraction

The relational abstraction is based on a standard transformation that replaces function symbols by relation symbols. While the standard transformation uses existential quantifiers, our relational abstraction uses universal quantifiers, thus eliminating edges from the sort-graph.

To express the relational abstraction of φ, we introduce fresh relation symbols corresponding to the function symbols in F:

Definition 4.1 (Relational Vocabulary). The *relational vocabulary* associated with F is $\mathcal{V}_{\alpha_F} = (\mathcal{V} \setminus F) \cup \{r_f \mid f \in F\}$, where for each $f \in F$ of sort $s_1 \times \cdots \times s_n \longrightarrow s_{n+1}$ (with $n \geq 1$), r_f is a fresh relation symbol of sort $s_1 \times \cdots \times s_n \times s_{n+1}$.

The intention of r_f is that $r_f(x_1, \dots, x_n, y)$ should hold exactly when $y = f(x_1, \dots, x_n)$. That is, r_f is meant to represent the graph of f. To create the relational abstraction, we replace every term t of the form $f(x_1, \dots, x_n)$ such that $f \in F$ by a unique free variable q_t and add a guard $r_f(x_1, \dots, x_n, q_t)$ that relates q_t with $f(x_1, \dots, x_n)$. Assuming r_f is a correct representation of the graph of f, the original formula holds exactly when the new formula holds for *all* values of the q_t variables such that $r_f(x_1, \dots, x_n, q_t)$. In fact, there is only one such value, so it matters not whether q_t is universally or existentially quantified. However, we prefer the universal quantifier in order to avoid a quantifier alternation. The formal definition of the encoding (Definition 4.2 below) is slightly more complicated since x_i, the arguments of f, may themselves be terms that include F symbols, thus introducing additional guards. In the definition, given a term t that may include F symbols, $\alpha_F^v(t)$ is the term that is used in place of t in the relational encoding (a fresh variable q_t as above, or t itself if it does not include F symbols), and $\alpha_F^g(t)$ is the guard formula used to enforce that $\alpha_F^v(t)$ represents t.

Definition 4.2 (Relational Encoding). For a given term t over $\mathcal{V}$, we define $\alpha_F^v(t)$ to be a term over $\mathcal{V}_{\alpha_F}$ and $\alpha_F^g(t)$ to be a conjunction of atomic formulas over $\mathcal{V}_{\alpha_F}$ such that:

- If t does not include any $f \in F$, then $\alpha_F^v(t) = t$ and $\alpha_F^g(t) = \textit{true}$.

- If $t = f(t_1, \ldots, t_n)$ for $f \in F$ of sort $s_1 \times \cdots \times s_n \longrightarrow s_{n+1}$, then $\alpha^v_F(t) = q_t$ and $\alpha^g_F(t) = r_f(\alpha^v_F(t_1), \ldots, \alpha^v_F(t_n), q_t) \wedge \bigwedge_{1 \le i \le n} \alpha^g_F(t_i)$.

Let φ be a closed formula in universal prenex form (in NNF). Then, the *relational encoding* of φ w.r.t. F, denoted $R_F(\varphi)$, is the universal closure of the formula obtained from φ when every literal $\ell(t_1, \ldots, t_k)$ is replaced by the formula

$$\left(\bigwedge_{i=1}^{k} \alpha^g_F(t_i) \right) \rightarrow \ell(\alpha^v_F(t_n), \ldots, \alpha^v_F(t_k)).$$

(Note that the transformation of the literals introduces fresh free variables that are universally quantified in $R_F(\varphi)$).)

Example 4.3. Equation (2) in Section 2.8 presents the relational encoding of the formula from Equation (1), where the universally-quantified variable N of sort node is introduced as an abstraction of the term $\mathit{inter}(Q_1, Q_2)$, and its use is guarded by the formula $r_{\mathit{inter}}(Q_1, Q_2, N)$.

The relational encoding on its own does not ensure that the newly introduced r_f symbols represent the graphs of functions. If we augment the relational encoding with axioms that ensure that the interpretation of r_f is in fact a function, then any model that satisfies $R_F(\varphi)$ induces a model that satisfies φ (and vice versa), simply by adopting the interpretation of r_f as the interpretation of the function symbol f. This is achieved by the following *functionality* and *totality* axioms.

Definition 4.4 (Functionality and Totality). For an $n{+}1$-ary relation symbol r of sort $s_1 \times \cdots \times s_n \times s_{n+1}$, we define the *functionality axiom* of r to be the formula $A_{\mathit{fun}}(r) = \forall x_1 : s_1, \ldots, x_n : s_n.\ \forall y_1 : s_{n+1}, y_2 : s_{n+1}.\ r(x_1, \ldots, x_n, y_1) \wedge r(x_1, \ldots, x_n, y_2) \rightarrow y_1 = y_2$. We define the *totality axiom* of r to be $A_{\mathit{tot}}(r) = \forall x_1 : s_1, \ldots, x_n : s_n.\ \exists y_1 : s_{n+1}.\ r(x_1, \ldots, x_n, y)$.

The functionality axiom states that for every tuple of elements in the "domain" of r there is at most one "image", and the totality axiom ensures that an "image" exists. Together, they ensure that r captures the graph of a function.

Theorem 4.5. *φ is satisfiable if and only if*

$$R_F(\varphi) \wedge \bigwedge_{f \in F} A_{\mathit{fun}}(r_f) \wedge \bigwedge_{f \in F} A_{\mathit{tot}}(r_f)$$

is satisfiable.

This holds because any model of φ may be transformed into a model of $\alpha_F(\varphi)$ by converting each function $f \in F$ to a corresponding relation r_f and *vice versa.*

The relational encoding is prenex universal, as are the functionality axioms. On the other hand, the totality axioms have a ∀∃ quantifier alternation. Indeed, the totality axioms introduce to the sort-graph the same arcs that the original functions in F do. Therefore, they are omitted from the relational abstraction, whose goal is to eliminate these edges:

Definition 4.6 (Relational Abstraction). The *relational abstraction* of φ w.r.t. F is a formula over $\mathcal{V}_{\alpha_F}$ defined by $\alpha_F(\varphi) = R_F(\varphi) \wedge \bigwedge_{f \in F} A_{\mathit{fun}}(r_f)$.

Note that the relational abstraction of φ is still a quantified formula. The following observation points out that relational abstraction achieves its primary goal of removing the arcs that correspond to the function symbols in F from the sort graph of φ, ensuring that the obtained formula is stratified.

Observation 1. *The sort graph of $\alpha_F(\varphi)$ is the same as the one induced by the function symbols in $\mathcal{V} \setminus F$.*

Theorem 4.7 (Soundness). *If $\alpha_F(\varphi)$ is unsatisfiable, then so is φ.*

Remark 1. *The relational encoding of a universal formula φ replaces a literal $\ell(t_1, \ldots, t_k)$ by the universal closure of $\left(\bigwedge_{i=1}^{k} \alpha_F^g(t_i)\right) \rightarrow \ell(\alpha_F^v(t_n), \ldots, \alpha_F^v(t_k))$, and pushes the universal quantifiers outwards (since all quantifiers in φ are also universal). Replacing $\ell(t_1, \ldots, t_k)$ by the existential closure of $\left(\bigwedge_{i=1}^{k} \alpha_F^g(t_i)\right) \wedge \ell(\alpha_F^v(t_n), \ldots, \alpha_F^v(t_k))$ is also sound (i.e., Theorems 4.5 and 4.7 still hold), and in fact results in an abstraction that is more precise (i.e., the obtained abstract formula implies $\alpha_F(\varphi)$). However, the existential quantifiers cannot be pushed outwards over the universal quantifiers in φ, and may induce new arcs in the sort graph. Still, for literals where all the arcs created by the existential quantifiers are already induced by the function symbols in $\mathcal{V} \setminus F$, we may use this more precise encoding in the definition of $\alpha_F(\varphi)$ without affecting the sort graph. This improves precision without compromising decidability.*

Soundness means that unsatisfiability is preserved from the abstraction to the original formula φ. This is a trivial consequence of Theorem 4.5. The opposite direction does not hold in general. For example, $\alpha_F(\varphi)$ may be satisfied by a model in which r_f is interpreted to be an empty relation, while φ is unsatisfiable, since no *function* satisfies it. As a concrete example, consider the following three invariants:

$$\begin{array}{ll} \text{[LeaderDecides]} & \forall T.\ \text{decision}(\text{leader}(T), T) \\ \text{[Settled]} & \forall T, N.\ \text{decision}(N, T) \rightarrow \text{settled}(T) \\ \text{[Safety]} & \forall T.\ \text{settled}(T) \end{array}$$

The first invariant states that the leader of time T decides in time T, the second states that if a node decides in time T then T is settled, and finally, the safety invariant states that every time T is settled. The first two invariants imply the third one (since the leader function ensures that every time has a leader). Using our terminology, this means that the formula obtained by taking the conjunction of the first two invariants with the negation of the third is unsatisfiable. However, this is no longer the case when the leader function is abstracted. Applying relational abstraction on the leader function changes the first invariant into:

$$\text{[LeaderDecides}'\text{]} \qquad \forall T, N.\ r_{\text{leader}}(T, N) \rightarrow \text{decision}(N, T)$$

Even with the addition of the functionality property of r_{leader}, a satisfying model that has no leader for some time T_0, and as a result, T_0 is not settled, emerges as a counterexample and the safety property cannot be verified.

4.2 Recovering Completeness via Instantiations

Theorem 4.5 ensures that if the totality axioms are included, then the relational abstraction preserves unsatisfiability of φ. Unfortunately, as explained above, the totality axioms re-introduce the sort-edges that correspond to the function in F—these are the very same function-edges that the relational abstraction was targeted at removing. Instead, we show that there always exists a finite set of instantiations of the totality axioms that suffice for preserving the unsatisfiability of the VCs. These instantiations refine the initial abstraction $\alpha_F(\varphi)$.

Each instantiation of a totality axiom asserts the existence of an image for a specific term in the domain of an abstracted function. Therefore, given any ground term t that includes the function symbols in F, we can use instantiations of the totality axioms to construct an existentially quantified variable q_t that corresponds to t in the relational encoding, and hence asserts its existence (q_t is later Skolemized, resulting in a Skolem constant). As an example, suppose that the proof of unsatisfiability relies on an element corresponding to $f(f(c))$, but no such element occurs in a given model of the relational abstraction (for example, because r_f is empty). We can force such an element to exist in the model by instantiating the totality axiom for f, first to obtain a variable $q_{f(c)}$ such that $r_f(c, q_{f(c)})$ holds and then to obtain a variable $q_{f(f(c))}$ such that $r_f(q_{f(c}, q_{f(f(c))})$ holds. The former variable captures $f(c)$, while the latter captures $f(f(c))$. In general, we construct the relational refinement by adding *relational instantiations* of terms t. These are defined as follows:

Definition 4.8 (Relational Instantiation). Given a ground term t over $\mathcal{V}$, the *relational instantiation* of t over $\mathcal{V}_{\alpha_F}$, denoted $inst_{\alpha_F}(t)$, is given by the existential closure of the formula $\alpha^g_F(t)$ defined in Definition 4.6.

Hence, $inst_{\alpha_F}(t)$ is a closed existentially-quantified formula over $\mathcal{V}_{\alpha_F}$ (and after Skolemization, it is a ground formula). The relational instantiation for t asserts, in effect, that there is some element in the model corresponding to t. Note that if t does not include any symbol from F, then $inst_{\alpha_F}(t) \equiv true$ (since in this case $\alpha^g_F(t) = true$, see Definition 4.6). That is, such instantiations do not refine the abstraction.

The relational instantiation $inst_{\alpha_F}(t)$ may be derived from the totality axioms by instantiating them recursively on the subterms of t, and then on t itself. We, therefore, have the following lemma:

Lemma 4.9. *For any ground term t over $\mathcal{V}$, we have that $\bigwedge_{f \in F} A_{tot}(r_f) \Longrightarrow inst_{\alpha_F}(t)$.*

This immediately implies that relational instantiations may soundly refine the relational abstraction:

Corollary 4.10 (Soundness of Refinement). *Let T be a finite set of ground terms over $\mathcal{V}$. If $\alpha_F(\varphi) \wedge \bigwedge_{t \in T} inst_{\alpha_F}(t)$ is unsatisfiable, so is φ.*

Importantly, refining the relational abstraction by adding instantiations does not introduce cycles between sorts, as it does not add any new arcs to the sort-graph of $\alpha_F(\varphi)$.

Observation 2. *If T is a finite set of ground terms over $\mathcal{V}$, then the sort graph of $\alpha_F(\varphi) \wedge \bigwedge_{t \in T} inst_{\alpha_F}(t)$ is the same as the sort graph of $\alpha_F(\varphi)$.*

Relational instantiations are also complete for refutation. That is, suppose that φ is unsatisfiable. By Herbrand's theorem and compactness, there exists a grounding of φ that is unsatisfiable, using a finite set of ground terms T (over $\mathcal{V}$). By augmenting $\alpha_F(\varphi)$, the relational abstraction of φ, with the relational instantiations for these terms T, we obtain a refined abstraction $\alpha_F(\varphi) \wedge \bigwedge_{t \in T} inst_{\alpha_F}(t)$ that is also unsatisfiable, since any model of the refinement can be converted to a model of the given grounding of φ. The idea is that the relational abstraction abstracts away the ground terms that include F symbols and as a result reduces the set of ground instances considered for φ. Relational instantiations, which instantiate the totality axioms, reintroduce ground terms that were abstracted and hence extend the set of ground instances that are considered in the context of $\alpha_F(\varphi)$. If φ is unsatisfiable, but $\alpha_F(\varphi)$ is satisfiable, then some of the ground instances of φ that are needed to establish unsatisfiability are missing due to the abstraction. Conjoining the relational abstraction $\alpha_F(\varphi)$ with relational instantiations has the effect of introducing the missing ground terms and corresponding ground instances. In fact, to achieve completeness, it suffices to consider relational instantiations for the subset of T that includes symbols from F. This is because F-free terms are not abstracted and can be used by the solver if needed — indeed, the relational instantiations for F-free terms are *true*.

Theorem 4.11 (Completeness of Refinement by Relational Instantiations). *If φ is unsatisfiable, then there exists a finite set T of ground terms over $\mathcal{V}$ s.t. the refined abstraction $\alpha_F(\varphi) \wedge \bigwedge_{t \in T} inst_{\alpha_F}(t)$ is unsatisfiable.*

Furthermore, as the following theorem implies, as long as the refined abstraction is still satisfiable, every satisfying model can be excluded by adding $inst_{\alpha_F}(t)$ for some term from the set T used to achieve completeness in Theorem 4.11. This means that an ideal user can always make progress in the refinement process.

Theorem 4.12 (Completeness of Refinement by Excluding Counterexamples). *Suppose that φ is unsatisfiable and let T be the set of ground terms over $\mathcal{V}$ from Theorem 4.11. Given a set T' of ground terms over $\mathcal{V}$, if $\alpha_F(\varphi) \land \bigwedge_{t \in T'} inst_{\alpha_F}(t)$ is satisfiable, and $m \models \alpha_F(\varphi) \land \bigwedge_{t \in T'} inst_{\alpha_F}(t)$, then there exists $t \in T$ such that $m \not\models inst_{\alpha_F}(t)$.*

We note that the relational abstraction of φ and its refinements are still quantified, which means that the user is not required to provide the full grounding of φ. Completing the task is left to the solver: assuming that F was chosen to eliminate all sort cycles, the abstraction and refinements are stratified, making it possible for the solver to exhaust all of their finitely many ground instances and determine unsatisfiability. Specifically, as commented above, the user need not instantiate the totality axioms on terms that do not include any F symbols (indeed, since such terms are not abstracted, they are always interpreted by the counterexamples provided by the solver, and their relational instantiations will never exclude the counterexamples).

5 PBFT AS AN EXAMPLE

This section demonstrates an instantiation step from identification to application on the PBFT protocol. PBFT operates in a Byzantine environment, which means that up to a third of the replicas in the protocol may be dishonest. A dishonest replica can send any message it wants but cannot masquerade as another—the protocol assumes that the sender of each message can be verified.

The critical safety invariant of PBFT is the agreement invariant:

$$[Agreement] \qquad (3)$$

$$\forall N_1, N_2 \colon \text{node}, I \colon \text{slot}, V_1, V_2 \colon \text{view}, O_1, O_2 \colon \text{value}.$$
$$\text{honest}(N_1) \land \text{honest}(N_2) \land \text{commit}(N_1, I, V_1, O_1) \land \text{commit}(N_2, I, V_2, O_2) \rightarrow O_1 = O_2$$

The agreement invariant is a crucial safety rule for any consensus protocol. It requires that if two honest replicas, represented by nodes N_1 and N_2, commit values, denoted as O_1 and O_2, at the same slot I but possibly in different views V_1 and V_2, then the committed operations must be identical.

PBFT ensures this agreement property by maintaining a variant of Lamport's choosable invariant. This invariant stipulates that whenever a new view V_2 begins, the primary (leader) of V_2 must propose value O_1 at slot I if it was already committed or could still be committed in an older view V_1 where $V_1 < V_2$. Honest replicas will not join the new view unless O_1 is proposed, as doing so could jeopardize the agreement property if a different value is decided upon.

Moreover, when an honest replica transitions to a new view, it sends a prepare message in response to each proposal from the primary.

The choosable invariant guarantees that if an honest replica (N_2) changes its view to V_2 but does not send a prepare message for (I, V_2, O) because the primary of V_2 proposed a different value than O, then O could not have been decided earlier. The invariant establishes this by stating that for every quorum Q of replicas, there is an honest replica N_3 that has already left view V_1 and did not send a commit message for (O, I, V_1). Therefore, no quorum Q of matched commit messages can be formed in V_1 for value O, and O cannot be committed in view V_1.

This limitation arises from the way commit (decide) operations function. To commit a value O at slot I in view V_1, an honest replica N_3 must receive a quorum of matching commit messages for (O, I, V_1). When combined with other invariants, the choosable invariant ensures that the agreement property is upheld every time a new value is committed in the decide action.

```
invariant[Agreement]
∀N1, N2: node, I: slot, V1, V2: view, O1, O2: value.
            commit(N1, I, V1, O1) ∧ commit(N2, I, V2, O2) → O1 = O2
instantiate for action decide
total[era](N1,V1) ∧ total[era](N2,V2)
```

Fig. 11. Instantiations of totality of r_{era}.

Once the log reaches its size limit, it is necessary to truncate it. After truncation, a replica no longer sends messages about slots older than its last checkpoint slot. As a result, the choosable invariant only applies to replicas where slot I is still valid, not to replicas whose last checkpoint is after I.

To account for this, we introduce a new (ghost) function called era. This function keeps track of the last checkpoint slot for each view, for each replica.

The choosable invariant is then:

$[Choosable]$

$$\forall V_1, V_2\colon \text{view},\ I\colon \text{slot}, O\colon \text{value}, Q\colon \text{quorum}. \Big(V_1 < V_2 \wedge \text{majority}(Q) \wedge$$

$$\exists N_2\colon \text{node}. \neg\text{prepareMsg}(N_2, I, V_2, O) \wedge \text{honest}(N_2) \wedge \text{newView}(N_2, V_2) \wedge \text{era}(N_2, V_2) \leq I\Big)$$

$$\rightarrow\ \exists N_3\colon \text{node}. \text{member}(N_3, Q) \wedge \text{honest}(N_3) \wedge \text{leftView}(N_3, V_1) \wedge \neg\text{commitMsg}(N_3, I, V_1, O)$$

Due to the multiple function cycles created by the era function, it is abstracted into a functional relation, denoted as r_{era}. The totality axiom, which introduces these function cycles, is hidden in this abstraction.

The abstraction re-formulates the choosable invariant using r_{era} (note the modifications in blue):

$[Choosable]$

$$\forall V_1, V_2\colon \text{view},\ I\colon \text{slot}, O\colon \text{value}, Q\colon \text{quorum}. \Big(V_1 < V_2 \wedge \text{majority}(Q) \wedge$$

$$\exists N_2\colon \text{node}, I_0\colon \text{slot}. \neg\text{prepareMsg}(N_2, I, V_2, O) \wedge \text{honest}(N_2) \wedge \text{newView}(N_2, V_2)$$

$$\wedge\ r_{\text{era}}(N_2, V_2, I_0) \wedge I_0 \leq I\Big)$$

$$\rightarrow\ \exists N_3\colon \text{node}. \text{member}(N_3, Q) \wedge \text{honest}(N_3) \wedge \text{leftView}(N_3, V_1) \wedge \neg\text{commitMsg}(N_3, I, V_1, O)$$

If we do not assume the totality of r_{era}, the choosable invariant alone cannot guarantee that the agreement property is preserved by the decide action. For example, assume by contradiction that N_2 has already committed O_2 at slot I in a view $V_2 > V_1$, and therefore did not send a prepare message for (I, V_1, O_1) and O_1 was decided in V_1. Without r_{era} being total, the choosable invariant will not be triggered on O_2 and will not result in a contradiction . In fact, a counterexample where N_2 does not have an era (r_{era} is empty) can be used to falsify the agreement invariant.

In order to complete the proof, the totality of r_{era} for two specific replicas and corresponding views are needed: (N_2, V_2) as above, and (N_1, V_1) defined similarly for the case where $V_2 < V_1$, where N_1, N_2, V_1, V_2 are the logical variables for which the agreement property is being established.

The instantiations are displayed in Figure 11. In the case of proving the agreement invariant only variables from the negation of the invariant were needed. As shown in the toy example in the overview, there might be a need for variables from the action itself. The diagnosability property of the methodology is maintained when using this style of instantiation. Using an irrelevant instantiation usually results in a similar counter example. In our PBFT model, there are no nested messages such as in [Castro and Liskov 2002]. For example, in a “new view” message, the modeling in [Castro and Liskov 2002] assumes the “new view” message contains all previous messages needed inside the message. In our “new view” message we look up the needed messages in the replica’s local storage.

5.1 A Full Counter Example

In this subsection, we aim to present a comprehensive counterexample that closely resembles the one obtained from Ivy. The emergence of this counterexample occurred during the incorporation of the *era* relation, outlined in detail in Section 5. The counterexample is relevant to the choosable invariant and arises during the verification of the "receive new view" action. For clarity, we have excluded all relations not present in the invariant and have focused solely on detailing the positive entries. This counterexample is inherently unreachable. It involves the scenario where the honest node n_1 commits to different values across two distinct views. This situation is feasible due to the absence of any content in the era relation within the prestate. As a result, the choosable invariant holds vacuously in this context.

6 EVALUATION

In order to determine the benefits of the methodology, we applied the methodology to many of the distributed consensus protocols. The goal is to determine the difficulty of formally verifying these protocols in our methodology and to compare it to the difficulty of using other methods. We determine the usefulness of using relational abstraction for proving complicated protocols. The question is how much can be saved by it and how a user addresses spurious failures. The goal is to determine: (i) What kind of protocols and properties can be verified using relational abstraction? (ii) When relational abstraction fails, can the user understand the reason behind the failure and overcome it? (iii) What protocols can be proven using the methodology?

6.1 Experimental Setup and Benchmarks

Table 1 summarizes the results of applying the methodology to Single decree Paxos and Multi Paxos [Lamport 2002a], Fast Paxos [Lamport 2002b], Flexible Paxos [Howard et al. 2016], Vertical Paxos [Lamport et al. 2009], Synchronized BFT [Abraham et al. 2020] and PBFT [Castro and Liskov 2002]. The Paxos variants assume fail-safe replica failure, while Synchronized BFT and PBFT assume Byzantine replica failure. Additionally, the Paxos variants and PBFT assume asynchronous message passing, whereas Synchronized BFT assumes synchronized message passing. Verification experiments were performed on a 3.00GHz 8-core i7-5960X CPU with 16GB of RAM machine. The SMT solver used is Z3 [de Moura and Bjørner 2008] version 4.6.1. We also tried verifying the protocols without abstractions and Z3 timed out in all cases (timeout was set to 5 hours). Note that while the verification time includes all inductiveness checks, a typical check is performed in less than a second depicted in column PR, and a typical counterexample is produced in less than 10 seconds (this includes minimizing it in terms of the number of elements).

6.2 Ineffectiveness of SMT Solvers on General Verification Conditions

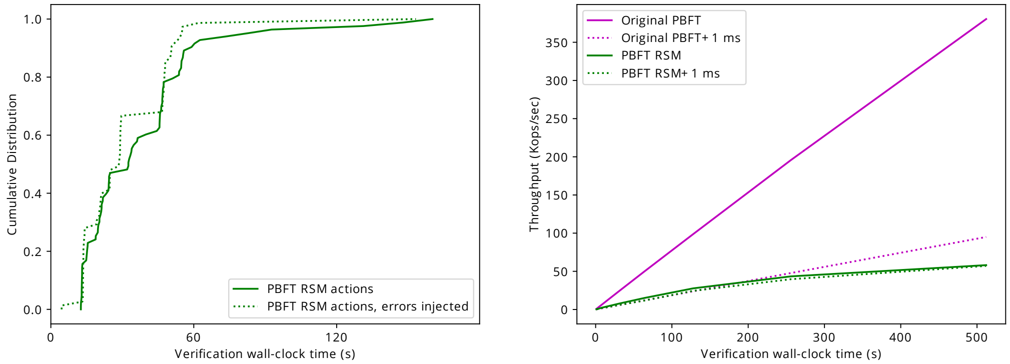
In our experiments, Z3 timeout on all the benchmarks formalized in FOL using the default E-Matching option in Z3. We also tried to evaluate other Z3 options. The Z3 SMT solver includes two options MBQI and E-matching. The E-Matching technique timed out (120 minutes) in 76 out of

```
Invariant:
$\forall VI_1, VI_2 : view, I : slot, Q : quourum, OP_1 : operation.$
$VI_2 > VI_1 \land$
$(\exists N_2 : replica, I0 : slot.$
$\neg$prepare_msg$(N_2, I, VI_2, OP_1) \land$is_good$(N2)$
$\land$has_new_view$(N_2, VI_2) \land$era$(N_2, VI_2, I_0) \land I_0 <= I) \rightarrow$
(majority$(Q) \rightarrow$
$(\exists N_3 : replica.N_3 \in Q \land$is_good$(N_3) \land$left_view$(N_3, VI_1) \land \neg$commit_msg$(N_3, I, VI_1, OP1))$
@Q = $q_0$
@I = $i_0$
@OP1 = $o_1$
@VI1 = $v_1$
@VI2 = $v_3$
**action** parameters:
quorum = $q_0$
node = $n_1$
primary = $n_0$
slot = $i_0$
view= $v_3$
prestate:
noneop = $o_1$
view.none = $v_0$
$v_0 < v_1 < v_2 < v_3$
$cr_0 = [i_0 : (v_2, o_0)]$
is_good$(n_1)$ = true
is_good$(n_0)$ = false
majority$(q_0)$ = true
$n_1 \in q_0$
prepare_msg$(n_0, i_0, v_3, o_0)$ = true
prepare_msg$(n_1, i_0, v_0, o_0)$ = true
prepare_msg$(n_1, i_0, v_1, o_1)$ = true
prepare_msg$(n_1, i_0, v_2, o_0)$ = true
commit_msg$(n_1, i_0, v_1, o_1)$ = true
commit_msg$(n_1, i_0, v_2, o_0)$ = true
view_change_msg$(n_1, v_3, cr_0)$ = true
has_new_view$(n_0, v_0)$ = true
has_new_view$(n_0, v_3)$ = true
has_new_view$(n_0, v_1)$ = true
has_new_view$(n_0, v_2)$ = true
has_new_view$(n_1, v_0)$ = true
has_new_view$(n_1, v_1)$ = true
has_new_view$(n_1, v_2)$ = true
left_view$(n_1, v_0)$ = true
left_view$(n_1, v_1)$ = true
left_view$(n_1, v_2)$ = true
post_state:
has_new_view$(n_1, v_3)$ = true
era$(n_1, v_3, i_0)$ = true
```

Fig. 12. The initial counterexample that emerged following the introduction of r_{era}.

Table 1. Applying our methodology to different protocols. LOC is the number of lines in the abstract protocol and CL is the number of lines of C++ for protocols for which we extracted the C++ implementation. # inv is the number of invariants in the final verification goal. # AF is the number of functions automatically abstracted into relations. # IC is the total number of inductiveness checks. # INS is the number of instantiations. SR is the percent of inductiveness checks automatically proven by the SMT solver after relational abstraction (without manual instantiations). PR is the percentage of inductive checks that are completed in less than 1 second (possibly with manual instantiations). TT is the total SMT time of the final verification step of the complete protocol. STD is the standard deviation in TT with 10 repeated runs. VT is the total SMT time for verifying the complete vanilla protocol expressed in FOL without any abstraction.

protocol	**LOC**	**CL**	**# inv**	**# AF**	**# IC**	**# INS**	**SR**	**PR**	**TT**	**STD**	**VT**
Single Paxos	220	n/a	12	7	60	1	98.3%	100%	1.5s	± 0.015s	>5h
Multi Paxos	259	n/a	14	9	84	1	98.8%	100%	3.3s	± 0.1s	>5h
Flexible Paxos	240	n/a	13	7	65	1	98.5%	100%	1.4s	± 0.03s	>5h
Fast Paxos	240	n/a	13	7	65	1	98.5%	100%	4.21s	± 0.073s	>5h
Vertical Paxos	362	n/a	23	7	207	2	99.03%	100%	4.78s	± 0.084s	>5h
SyncBFT	179	n/a	7	9	36	0	100%	100%	2.1s	± 0.05s	>5h
PBFT RSM	1535	8k	73	40	3562	6	99.9%	99.9%	57m	± 15.4s	>5h
PBFT RSM+hash	1632	8.1k	75	44	4231	8	99.9%	99.9%	59m	± 13.7s	>5h
PhbBFT	1712	8.1k	75	49	4838	8	99.9%	99.9%	59m	± 5.7s	>5h



(a) Verification times are interactive and predictable.

(b) Throughput of verified vs unverified.

Fig. 13. Throughput and verification times

84 verification conditions while MBQI timed out in 48 out of 84 verification conditions. The VCs origin is FOL model of Multi Paxos.

6.3 Scalability and Proof Burden

Verification frameworks often look elegant on small examples but generalize poorly as they are applied to more complex systems. Our evaluation shows that our methodology is applicable to complex consensus protocols with a modest human effort. In Single decree Paxos, Multi Paxos , Fast Paxos and Flexible Paxos examples, most of the VCs were discharged automatically by the SMT solver, where the user needed to add only one instantiation. This instantiation is required since the totality property of the function in the promise message (1b) is abstracted away. SyncBFT agreement property was verified without any need to use instantiations.

For all of the above benchmarks, we verified the safety invariant of the protocol design. To demonstrate the scalability of this approach, we verified a variant of the PBFT protocol end-to-end down to a generated C++ code. In the variant we verified, taken from [Castro and Liskov 2002], messages are asymmetrically signed [Courtois et al. 2003] by each replica and not the variant that uses MACs. Naturally, the PBFT model is bigger than the previous protocols due to its complexity and the need to concertize every replica state in order to allow code generation directly from the model. Moreover, our PBFT model includes state transfer. We proved the state machine replication safety property, i.e., every two honest replicas that are in the same position in the log, have the same state. The level of detail in the model used for extraction closely resembles the corresponding C++ code. However, there are notable differences in the way Ivy handles certain statements. For example, Ivy allows the use of statements like $x := *$ to indicate that variable x can take any value, followed by *assume* $\phi(x)$ statements for model verification purposes. When extracting the model to code, it becomes necessary to determine the specific value that x takes on. In such cases, the user typically introduces a function that computes x's postcondition, ensuring it satisfies phi. This verification step is performed by Ivy. It is important to note that the C++ extractor does not introduce any optimizations beyond what is explicitly captured in the model itself. In order to show the extensibility and reusability of the methodology, we also verified two more optimizations on top of the first model. The first optimization uses hash in some messages to reduce message size and was also implemented in the original implementation. The second optimization was not implemented previously and changes the protocol message flow. This optimization reuses messages from the state transfer protocol to discard commit messages from the original protocol. Interestingly, both optimizations reused the original invariant and instantiations with minor changes. Both optimizations required strengthening the invariant.

We find these results encouraging since most of the VCs, i.e., 99% were automatically discharged by the SMT prover. Another encouraging fact is the stability of the SMT solvers w.r.t., repeated runs measured by the standard deviation across different runs. The amount of manual work in instantiations is also reasonable. Finding the right instantiation blends naturally in the workflow. A counterexample is generated by the SMT solver and the user tracks down the missing piece. Usually, when adding a useless instantiation, the next counterexample will be very similar. Fortunately, even for big systems like PBFT and end-to-end verification 6 instantiations suffice to complete the proof. Moreover, similar variants of the same protocol, Paxos for example, require small modifications in the instantiations if any. The instantiation used in Single decree Paxos, Multi Paxos, Fast Paxos, and Flexible Paxos is almost identical.

Our implementation of PBFT Replica State Machine has 744 lines of imperative code; the extracted C++ program has 5k lines of code and 2k lines of code of the trusted codebase implementing general utilities such as data structures. Castro's unverified implementation is 20k of lines of C. The Castro's implementation didn't rely on external libraries and consists of code that implements additional functionality besides consensus like message delivery, and cryptography. In contrast, our implementation relies on trusted libraries, which accounts for the difference in lines of code. Velisarios PBFT [Rahli et al. 2018] is verified in Coq. It consists of 20k lines of imperative code and 22k lines of proofs for a total of 44k lines of code. That is a much more complex verification project than our PBFT implementation, which has 1.6k total lines of code.

6.4 Development Process Predictability

The SMT solver verifies our complete PBFT Replica State Machine implementation in 57m. In ordinary use, the developer modifies a single action and uses SMT to determine if the invariant is inductive, or if she modifies an invariant when a CTI is provided. The Figure 13a compares the verification time of correct actions and actions with injected errors. The solid green line plots the independent verification times of the 84 actions in our PBFT RSM; checking a single action rarely takes more than a minute. These are reasonable interactive verification times; furthermore, we observe that these times are representative not only of the correct system but of the system during the development process. We simulated this situation by injecting errors into actions by commenting out randomly-selected lines until the action failed to maintain its module's invariants. This technique yielded 76 synthetically broken actions shown as a dotted green line Figure 13a (other actions broke the system in ways not visible to this experiment).

6.5 Comparing to Previous Decidable Logic Models

The ability to use general and not stratified logic drastically improves the verification power. Our verification of Multi Paxos requires 259 lines of code and 1 instantiation vs. [Padon et al. 2017b] which requires 445 lines of code and several non-trivial transformations on verification conditions. Single decree Paxos, Flexible Paxos, and Fast Paxos required similar several non-trivial transformations on verification conditions in [Padon et al. 2017b] while we needed a single instantiation for each protocol. While modeling Vertical Paxos we noticed and confirmed with the author that the model in [Padon et al. 2017b] is not in a decidable fragment. Our verification of Vertical Paxos uses only decidable SMT queries and requires 2 instantiations and no complicated transformations. To our knowledge, our model of Vertical Paxos is the first model in decidable logic.

6.6 Evaluating the Performance of the Extracted C++ Implementation

We used machines with 4 VCPUs, Intel Xeon Platinum 8000 Series (Skylake-SP) processors with clock speed 3.1 GHz and 32 GiB RAM. Machines run Ubuntu 16.04.2 LTS (64 bit) and are connected via a 10 Gb network. The median ping between machines is 0.91ms, and the average is 0.53ms. The network behavior was modulated by Linux Traffic Control. The state machine is a key-value store, and the workload is 50% writes uniformly distributed across a space of 1,000 keys. Value sizes range from 4B to 4MB. Asymmetric signatures use 2048-bit RSA keys. The PBFT implementation is updated to use SHA256 instead of MD5. Figure 13b shows the throughput of PBFT RSM at various batch sizes, as compared with the original Castro and Liskov's PBFT implementation (higher is better). At datacenter rates, Our implementation lags due to the cost of digital signatures not hidden behind network latency. At large batch sizes, our implementation lags due to the absence of digests. But with a modest latency (+1 ms), and for batch sizes below 200, PBFT RSM offers competitive performance.

7 RELATED WORK

7.1 Deductive Verification in Decideable Logic

[Taube et al. 2018] requires that verification conditions reside in a known decidable fragment. To achieve this, the user exploits modularity to break the verification condition into several verification conditions, each of which has no function cycles. This is a form of rely-guarantee, proving the verification condition of one module while relying on the properties of other modules. Unfortunately, modularity is not a panacea. Restricting the code to leverage modularity for decidability is hard to scale. The main challenge in this technique is to find the right partition of modules and invariants across them such that every sub-module emits a decidable VC and that all the VCs are discharged. Even when the problem is divided into modules, the sub-problems may still not fall into the decidable fragment. The modularity methodology is lacking in guidance on the next useful step the user should take when function cycles still exist. In fact, it is not clear if modularity can always be used. For example, log truncation is a natural optimization implemented in realistic consensus protocols that many realistic verification efforts ignore [Taube et al. 2018; Wilcox et al. 2015]. We have failed to reason about log-truncation using modularity (see Section 6). Derived relations can also be used to simplify the verification conditions to be expressible in a decidable logic [Padon et al. 2017a]. We extensively tried to use these methodologies to verify the PBFT protocol and could not find a partition that allows using the modularity methodology nor derived relations that eliminate all function cycles. For the Paxos protocol, our methodology leads to a simpler specification.

7.2 Applying SMT for Verifying Distributed Systems

In the IronFleet project, Dafny [Leino 2017] was successfully used to verify both the safety and liveness of distributed protocols [Hawblitzel et al. 2017]. The use of powerful first-order logic simplifies the specification but makes SMT reasoning hard and thus requires tricky user tuning using triggers. As observed in [Leino and Pit-Claudel 2016] and the IronFleet project, triggers are unstable over changes, leading to proofs that are sensitive to minor changes (referred as a “butterfly effect”).

7.3 Quantifier Instantiation

The use of instantiation in first-order logic goes back to Herbrand [Buss 1995] and Natural Deduction [Aschieri and Zorzi 2016]. Automatic quantifier instantiation is an active area of research with competing approaches, including [Bansal et al. 2015; Ge and de Moura 2009; Niemetz et al. 2021]. An interesting way to visualize quantifier instantiation was proposed in [Becker et al. 2019]. Visualization can be effective for E-matching-based quantifier instantiation and for manually adding triggers. Our approach relies on the use of decidable logic for which Z3 is a decision procedure, and allows the user to manually specify quantifier instantiations directly (as opposed to indirectly through triggers). Our results are inspired by recent results in [Löding et al. 2018] which shows the completeness of instantiation for natural proofs.

7.4 Using Proof Assistants

The Coq proof assistant was successfully used in Verdi [Wilcox et al. 2015] to prove the correctness of the Raft protocol [Ongaro and Ousterhout 2014] and to verify the PBFT protocol in [Rahli et al. 2018]. However, such proofs are laborious. For example, [Wilcox et al. 2015] verifies 300 lines of the simplified Raft consensus protocol using 50000 complex lines of Coq proof script written in 3.7 person-years. It also ignores reconfiguration and log truncation. Decomposing programs and verification conditions can simplify the verification task, e.g. [Andersen and Sergey 2021; Sergey et al. 2018]. Another approach to simplify the verification task is by providing a language based support for specification, e.g., [Desai et al. 2018; Lesani et al. 2016; Rahli et al. 2017]. This can be realized in proof assistants such as NuPurl [Allen et al. 2000]. In contrast to these efforts, our approach utilizes decision procedures implemented in SMT solvers for finding proofs and concrete counterexamples.

7.5 Decidable Infinite State Verification

An alternative approach to verifying distributed systems is specifying the safety and liveness properties of the system in a restricted manner and either showing a small bound property or developing specialized decision procedures. Two elegant approaches are presented in [Dragoi et al. 2016; Konnov et al. 2017a]. This automates the verification phase and enables proving liveness properties. Unfortunately, protocols such as PBFT are beyond reach for such approaches.

7.6 Finite State Model Checking

Finite state model checking has been used to verify finite state and parameterized distributed protocols, e.g., [Ashmore et al. 2019; Bokor et al. 2011; Jackson 2019; Konnov et al. 2017b; Lamport 2002c]. These techniques usually reason about designs and not actual implementations. We are interested in reasoning all the way from the design to an efficient implementation of a distributed protocol. For example, most bugs occurred in the implementation of the PBFT [Castro and Liskov 2002] protocol whose design has been checked [Castro and Liskov 1999].

7.7 Counter-Example-Guided Abstraction-Refinement

Counterexample-guided abstraction refinement(CEGAR), e.g., [Ball et al. 2001; Clarke et al. 2000; Henzinger et al. 2004], is a useful technique for automatically refining proofs based on counterexamples. Our can be viewed as an instance of the CEGAR framework. However, instead of representing the (finite) abstract transition system using Boolean formulas, our abstract transition systems are still infinite-state and we use first order logic formulas with quantifiers to encode them. We let the user guide instantiations when relational abstractions fail. Fully automating this approach remains a topic for further research. The main challenge is the exponential nature of choosing the right subset of functions to abstract (in our example we have 40+ function symbols).

8 CONCLUSION

We described an effective methodology for harnessing SMT solvers for verifying infinite state systems. It interacts with SMT using decidable queries and utilizes relational abstraction for eliminating quantifier alternation cycles. It also allows manual quantifier instantiations guided counterexamples. We have applied the methodology to verify designs and implementations of consensus protocols. Our attempt is to reach a good balance between proof-automation and expressiveness, without scarifying diagonalizability. The use of decidable logic provides stability and predictability for SMT solvers. Relational abstraction allows the user to express natural invariants and handles most of the verification goals automatically. The remaining goals are handled by the user using manual instantiations. We showed that this can be done with a moderate effort.

ACKNOWLEDGMENTS

The research leading to these results has received funding from the European Research Council under the European Union’s Horizon 2020 research and innovation programme (grant agreement No [759102-SVIS]). This research was partially supported by the Israeli Science Foundation (ISF) grant No. 1810/18.

REFERENCES

Ittai Abraham, Dahlia Malkhi, Kartik Nayak, Ling Ren, and Maofan Yin. 2020. Sync HotStuff: Simple and Practical Synchronous State Machine Replication. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020.* IEEE, 106–118.

Stuart F. Allen, Robert L. Constable, Richard Eaton, Christoph Kreitz, and Lori Lorigo. 2000. The Nuprl Open Logical Environment. In *Automated Deduction - CADE-17, 17th International Conference on Automated Deduction, Pittsburgh, PA, USA, June 17-20, 2000, Proceedings (Lecture Notes in Computer Science, Vol. 1831)*, David A. McAllester (Ed.). Springer, 170–176.

Kristoffer Just Arndal Andersen and Ilya Sergey. 2021. Protocol combinators for modeling, testing, and execution of distributed systems. *J. Funct. Program.* 31 (2021), e3.

Federico Aschieri and Margherita Zorzi. 2016. On natural deduction in classical first-order logic: Curry-Howard correspondence, strong normalization and Herbrand’s theorem. *Theor. Comput. Sci.* 625 (2016), 125–146. https://doi.org/10.1016/j.tcs.2016.02.028

Rylo Ashmore, Arie Gurfinkel, and Richard J. Trefler. 2019. Local Reasoning for Parameterized First Order Protocols. In *NASA Formal Methods - 11th International Symposium, NFM 2019, Houston, TX, USA, May 7-9, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11460)*, Julia M. Badger and Kristin Yvonne Rozier (Eds.). Springer, 36–53.

Thomas Ball, Rupak Majumdar, Todd D. Millstein, and Sriram K. Rajamani. 2001. Automatic Predicate Abstraction of C Programs. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Snowbird, Utah, USA, June 20-22, 2001*, Michael Burke and Mary Lou Soffa (Eds.). ACM, 203–213. https://doi.org/10.1145/378795.378846

Kshitij Bansal, Andrew Reynolds, Tim King, Clark W. Barrett, and Thomas Wies. 2015. Deciding Local Theory Extensions via E-matching. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 9207)*, Daniel Kroening and Corina S. Pasareanu (Eds.). Springer, 87–105.

Nils Becker, Peter Müller, and Alexander J. Summers. 2019. The Axiom Profiler: Understanding and Debugging SMT Quantifier Instantiations. In *Tools and Algorithms for the Construction and Analysis of Systems - 25th International Conference, TACAS 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 11427)*, Tomás Vojnar and Lijun Zhang (Eds.). Springer, 99–116. https://doi.org/10.1007/978-3-030-17462-0_6

Péter Bokor, Johannes Kinder, Marco Serafini, and Neeraj Suri. 2011. Efficient model checking of fault-tolerant distributed protocols. In *Proceedings of the 2011 IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2011, Hong Kong, China, June 27-30 2011*. IEEE Compute Society, 73–84.

Samuel R. Buss. 1995. On Herbrand's Theorem. *In Logic and Computational Complexity, Lecture Notes in Computer Science* 960 (1995), 195–209.

Miguel Castro and Barbara Liskov. 1999. . Technical Report. MIT. https://www.microsoft.com/en-us/research/wp-content/uploads/2017/01/tm590.pdf.

Miguel Castro and Barbara Liskov. 2002. Practical byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.* 20, 4 (2002), 398–461.

Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. 2000. Counterexample-Guided Abstraction Refinement. In *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings (Lecture Notes in Computer Science, Vol. 1855)*, E. Allen Emerson and A. Prasad Sistla (Eds.). Springer, 154–169. https://doi.org/10.1007/10722167_15

Nicolas T. Courtois, Louis Goubin, and Jacques Patarin. 2003. SFLASHv3, a fast asymmetric signature scheme. *IACR Cryptol. ePrint Arch.* (2003), 211. http://eprint.iacr.org/2003/211

Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS*. 337–340.

Ankush Desai, Amar Phanishayee, Shaz Qadeer, and Sanjit A. Seshia. 2018. Compositional programming and testing of dynamic distributed systems. *Proc. ACM Program. Lang.* 2, OOPSLA (2018), 159:1–159:30.

Cezara Dragoi, Thomas A. Henzinger, and Damien Zufferey. 2016. PSync: a partially synchronous language for fault-tolerant distributed algorithms. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, Rastislav Bodík and Rupak Majumdar (Eds.). ACM, 400–415.

Yeting Ge and Leonardo de Moura. 2009. Complete Instantiation for Quantified Formulas in Satisfiability Modulo Theories. In *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5643)*. Springer, 306–320.

Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath T. V. Setty, and Brian Zill. 2017. IronFleet: proving safety and liveness of practical distributed systems. *Commun. ACM* 60, 7 (2017), 83–92.

Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L. McMillan. 2004. Abstractions from proofs. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*, Neil D. Jones and Xavier Leroy (Eds.). ACM, 232–244. https://doi.org/10.1145/964001.964021

Heidi Howard, Dahlia Malkhi, and Alexander Spiegelman. 2016. Flexible Paxos: Quorum intersection revisited. *CoRR* abs/1608.06696 (2016). arXiv:1608.06696 http://arxiv.org/abs/1608.06696

Daniel Jackson. 2019. Alloy: a language and tool for exploring software designs. *Commun. ACM* 62, 9 (2019), 66–76.

Igor V. Konnov, Marijana Lazic, Helmut Veith, and Josef Widder. 2017a. A short counterexample property for safety and liveness verification of fault-tolerant distributed algorithms. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 719–734.

Igor V. Konnov, Marijana Lazic, Helmut Veith, and Josef Widder. 2017b. A short counterexample property for safety and liveness verification of fault-tolerant distributed algorithms. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D.

Gordon (Eds.). ACM, 719–734.

Leslie Lamport. 2002a. Paxos Made Simple, Fast, and Byzantine. In *Procedings of the 6th International Conference on Principles of Distributed Systems. OPODIS 2002, Reims, France, December 11-13, 2002 (Studia Informatica Universalis, Vol. 3)*, Alain Bui and Hacène Fouchal (Eds.). Suger, Saint-Denis, rue Catulienne, France, 7–9.

Leslie Lamport. 2002b. Paxos Made Simple, Fast, and Byzantine. In *Procedings of the 6th International Conference on Principles of Distributed Systems. OPODIS 2002, Reims, France, December 11-13, 2002 (Studia Informatica Universalis, Vol. 3)*, Alain Bui and Hacène Fouchal (Eds.). Suger, Saint-Denis, rue Catulienne, France, 7–9.

Leslie Lamport. 2002c. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers.* Addison-Wesley.

Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. 2009. Vertical paxos and primary-backup replication. In *Proceedings of the 28th Annual ACM Symposium on Principles of Distributed Computing, PODC 2009, Calgary, Alberta, Canada, August 10-12, 2009*, Srikanta Tirthapura and Lorenzo Alvisi (Eds.). ACM, 312–313. https://doi.org/10.1145/1582716.1582783

K. Rustan M. Leino. 2017. Accessible Software Verification with Dafny. *IEEE Software* 34, 6 (2017), 94–97.

K. Rustan M. Leino and Clément Pit-Claudel. 2016. Trigger Selection Strategies to Stabilize Program Verifiers. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 9779)*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer, 361–381.

Mohsen Lesani, Christian J. Bell, and Adam Chlipala. 2016. Chapar: Certified Causally Consistent Distributed Key-Value Stores. In *POPL'16: Proceedings of the 43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg, FL, USA). http://adam.chlipala.net/papers/ChaparPOPL16/

Christof Löding, P. Madhusudan, and Lucas Peña. 2018. Foundations for natural proofs and quantifier instantiation. *Proc. ACM Program. Lang.* 2, POPL (2018), 10:1–10:30.

Kenneth McMillan. 2020. Ivy. (2020).

Aina Niemetz, Mathias Preiner, Andrew Reynolds, Clark W. Barrett, and Cesare Tinelli. 2021. Syntax-Guided Quantifier Instantiation. In *Tools and Algorithms for the Construction and Analysis of Systems - 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 12652)*, Jan Friso Groote and Kim Guldstrand Larsen (Eds.). Springer, 145–163.

Diego Ongaro and John K. Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014*, Garth Gibson and Nickolai Zeldovich (Eds.). USENIX Association, 305–319.

Oded Padon, Giuliano Losa, Mooly Sagiv, and Sharon Shoham. 2017a. Paxos made EPR: decidable reasoning about distributed protocols. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 108:1–108:31.

Oded Padon, Giuliano Losa, Mooly Sagiv, and Sharon Shoham. 2017b. Paxos made EPR: decidable reasoning about distributed protocols. *PACMPL* 1, OOPSLA (2017), 108:1–108:31.

Oded Padon, Kenneth L. McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. 2016. Ivy: safety verification by interactive generalization. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, Chandra Krintz and Emery Berger (Eds.). ACM, 614–630.

Vincent Rahli, David Guaspari, Mark Bickford, and Robert L. Constable. 2017. EventML: Specification, verification, and implementation of crash-tolerant state machine replication systems. *Sci. Comput. Program.* 148 (2017), 26–48.

Vincent Rahli, Ivana Vukotic, Marcus Völp, and Paulo Jorge Esteves Veríssimo. 2018. Velisarios: Byzantine Fault-Tolerant Protocols Powered by Coq. In *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings.* 619–650.

Ilya Sergey, James R. Wilcox, and Zachary Tatlock. 2018. Programming and proving with distributed protocols. *Proc. ACM Program. Lang.* 2, POPL (2018), 28:1–28:30.

Marcelo Taube, Giuliano Losa, Kenneth L. McMillan, Oded Padon, Mooly Sagiv, Sharon Shoham, James R. Wilcox, and Doug Woos. 2018. Modularity for decidability of deductive verification with applications to distributed systems. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018.* 662–677.

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

Received 2023-04-14; accepted 2023-08-27