
# Accelerating Protocol Synthesis and Detecting Unrealizability with Interpretation Reduction

Derek Egolf(✉) and Stavros Tripakis

Northeastern University, Boston, MA, USA
{egolf.d,stavros}@northeastern.edu

**Abstract.** We present a novel counterexample-guided, sketch-based method for the synthesis of symbolic distributed protocols in TLA$^+$. Our method's chief novelty lies in a new search space reduction technique called interpretation reduction, which allows to not only eliminate incorrect candidate protocols before they are sent to the verifier, but also to avoid enumerating redundant candidates in the first place. Further performance improvements are achieved by an advanced technique for exact generalization of counterexamples. Experiments on a set of established benchmarks show that our tool is almost always faster than the state of the art, often by orders of magnitude, and was also able to synthesize an entire TLA$^+$ protocol "from scratch" in less than 3 minutes where the state of the art timed out after an hour. Our method is sound, complete, and guaranteed to terminate on unrealizable synthesis instances under common assumptions which hold in all our benchmarks.

## 1 Introduction

Distributed protocols are the foundation of many modern computer systems. They find use in domains such as finance [8,7] and cloud computing [9,10]. Distributed protocols are notoriously difficult to design and verify. These difficulties have motivated the advances in fuller automation for verification of both safety [18,20,45,43,37,36] and liveness [44] properties of distributed protocols.

In this paper, we study the automated *synthesis* of distributed protocols. In verification, the goal is to check whether a given protocol satisfies a given property. The goal in synthesis is to *invent* a protocol that satisfies a given property. Synthesis is a challenging problem [33,25,34,40,41], and we use the *sketching* paradigm [38,39] to turn the synthesis problem into a search problem. The user provides a protocol *sketch*, which is a protocol with holes in it; each hole is associated with a grammar. The goal is to choose expressions from the grammars to fill the holes in the sketch to obtain a protocol that satisfies a given property. Our target protocols are represented symbolically in the TLA$^+$ language [28], a popular specification language for distributed systems used in both academia and industry [32].

In this paper we propose a novel method for synthesis of distributed protocols by sketching. Our method follows the *counterexample-guided inductive synthesis* (CEGIS) paradigm [39,19], but with some crucial innovations, explained below.


A. Gurfinkel and M. Heule (Eds.): TACAS 2025, LNCS 15697, pp. 155–176, 2025.
https://doi.org/10.1007/978-3-031-90653-4_8

In CEGIS, a *learner* is responsible for generating candidate protocols, and a *verifier* is responsible for checking whether the candidate protocols satisfy the property. In addition to saying correct/incorrect, the verifier also provides a counterexample when the candidate protocol is incorrect. Existing approaches *generalize* the counterexample to a set of constraints which allows to *prune* the search space [2,4,12]. Such pruning eliminates candidate protocols that are sure to exhibit previously encountered counterexamples before they make it to the verifier, which is crucial for scalability: e.g., [12] show that pruning constraints can be used to prevent hundreds of thousands of model-checker calls.

Unfortunately, pruning alone only goes so far. Even when many calls to the verifier are avoided, the sheer size of the search space is often huge. Therefore, enumerating all candidates (and checking each one of them against the pruning constraints) can be prohibitive in itself.

In this paper, we address this problem by introducing a novel search space reduction technique called *interpretation reduction*. While pruning constraints eliminate incorrect candidate protocols before they are sent to the verifier, reduction avoids enumerating redundant candidates in the first place. Reduction can be achieved by choosing an equivalence relation, partitioning the search space into classes of equivalent expressions, and enumerating at most one expression from each equivalence class. One choice of equivalence relation is *universal equivalence*. Two expressions are universally equivalent if they evaluate to the same value under all interpretations (an *interpretation* is an assignment mapping terminal symbols of sketch grammars to values). E.g., $x+y$ and $y+x$ are universally equivalent. Our expression enumerator does better, and will only generate semantically distinct expressions up to a coarser notion of equivalence, namely, *interpretation equivalence*. If $\alpha$ is an interpretation, then two expressions, $e_1$ and $e_2$, are interpretation equivalent with respect to $\alpha$ if they evaluate to the same value under $\alpha$. E.g., $x$ and $x + x$ are not universally equivalent, but are interpretation equivalent with respect to the interpretation $\alpha = [x \mapsto 0]$. Our experiments show that our method is almost always faster than the state of the art, and in many cases, more than 100 times faster.

In addition to improving efficiency, interpretation reduction also enables recognizing *unrealizable* synthesis instances, i.e., instances where there is no solution. Under certain conditions (met in all of our benchmarks), interpretation reduction partitions the search space into finitely many equivalence classes. Then, if the algorithm has enumerated an expression from each equivalence class and none of them satisfy the property, the algorithm can infer that the synthesis instance is unrealizable and terminate. Our experiments show that our method is able to recognize five times more unrealizable synthesis instances than the state of the art.

In summary, this work makes the following contributions: (1) a novel search space reduction technique based on interpretation equivalence, (2) the first, to our knowledge, method for synthesizing TLA$^+$ protocols by sketching that is guaranteed to terminate on unrealizable synthesis instances when there are a finite number of equivalence classes modulo interpretation equivalence, and (3) a

synthesis tool called PolySemist that implements our method and outperforms the state of the art. Most notably, PolySemist was able to synthesize an entire TLA$^+$ protocol from scratch in less than 3 minutes where the state of the art timed out after an hour. To the best of our knowledge, ours is the first tool to synthesize a TLA$^+$ protocol from scratch.

## 2 Distributed Protocol Synthesis

**Protocol Representation in TLA$^+$** We consider symbolic transition systems modeled in TLA$^+$ [28], e.g., as shown in Fig. 1. A primed variable, e.g., *vote_ yes*$'$, denotes the value of the variable in the next state. Formally, a *protocol* is a tuple $\langle$Vars, Init, Next$\rangle$. Vars is the set of *state variables* (e.g. Fig. 1 line 2). Init and Next are predicates specifying, respectively, the *initial states* and the *transition relation* of the system, as explained in detail in the next subsections.

In Fig. 1, the protocol is parameterized by the set of nodes participating in the protocol, which is denoted by the declaration on line 1. As in [12], we are able to synthesize parameterized protocols. For space reasons, we omit all discussion of parameterized protocols in this paper and refer the reader to [12] for details.

**Protocol Semantics** A *state* of a protocol is an assignment of values to the variables in Vars. We write $s[v]$ to denote the value of the state variable $v$ in state $s$. Init is a predicate mapping a state to true or false; if a state satisfies Init (if it maps to true), it is an initial state of the protocol.

The transition relation Next is a predicate mapping a pair of states to true or false. If a pair of states $(s, t)$ satisfies Next, then there is a *transition* from $s$ to $t$, and we write $s \to t$. A state is *reachable* if there exists a *run* of the protocol containing that state. A run of a protocol is a possibly infinite sequence of states $s_0, s_1, s_2...$ such that (1) $s_0$ satifies Init, (2) $s_i \to s_{i+1}$ for all $i \geq 0$, and (3) the sequence satisfies optional *fairness constraints*. We omit a detailed discussion of fairness, but it is used to exclude runs where a transition is never taken, even though it was enabled infinitely often.

**Properties and Verification** We support standard temporal safety and liveness *properties* for specifying protocol correctness. Safety is often specified using a state *invariant*: a predicate mapping a state to true or false. A protocol satisfies a state invariant if all reachable states satisfy the invariant. A protocol satisfies a temporal property if all runs (or fair runs, if fairness is assumed) satisfy the property.

**Modeling Conventions** We adopt standard conventions on the syntax used to represent protocols, particularly on how Next is written. Specifically, we decompose Next into a disjunction of *actions* (e.g. Fig. 1 lines 15-17). An action is a predicate mapping a pair of states to true or false; e.g., action *GoCommit* of Fig. 1. We decompose an action into the conjunction of a *pre-condition* and

```
1  CONSTANT Node
2  vars := (vote_yes, go_commit, go_abort)
3  GoCommit :=
4        ∧ vote_yes = Node
5        ∧ go_commit' = Node
6        ∧ go_abort' = go_abort
7  VoteYes(n) :=
8        ∧ vote_yes' = vote_yes ∪ {n}
9        ∧ go_commit' = go_commit
10       ∧ go_abort' = go_abort
11 Init :=
12       ∧ vote_yes = ∅
13       ∧ go_commit = ∅
14       ∧ go_abort = ∅
15 Next :=
16       ∨ GoCommit
17       ∨ ∃n ∈ Node : VoteYes(n)
```

**Fig. 1.** An example of a TLA$^+$ protocol (excerpt).

a *post-condition*. A pre-condition is a predicate mapping a state to true or false; if the pre-condition of an action is satisfied by a state, then we say the action is *enabled* at that state. For instance, Fig. 1 line 4 says that action *GoCommit* is enabled only when all nodes have voted yes.

We decompose a post-condition into a conjunction of *post-clauses*, one for each state variable. A post-clause determines how its associated state variable changes when the action is taken. For instance, Fig. 1 line 5 shows a post-clause for the state variable *go_commit*, denoted by priming the variable name: *go_commit*$'$.

If $s \to t$ is a transition and $(s, t)$ satisfies an action $A$, we can say that $A$ *is taken* and write $s \xrightarrow{A} t$. Note that $(s, t)$ may satisfy multiple actions and we may annotate the transition with any of them. In this way, runs of a protocol may be outfitted with a sequence of actions. Annotating runs of a protocol with actions is critical for our synthesis algorithm, since annotations allow us to "blame" particular actions for causing a counterexample run.

**Protocol Synthesis** A tuple $\langle \textsc{Vars}, \textsc{Holes}, \textsc{Init}, \textsc{Next}_0 \rangle$ is a *protocol sketch*, where Vars and Init are as in a TLA$^+$ protocol and $\textsc{Next}_0$ is a transition relation predicate containing the hole names found in Holes. Holes is a finite (possibly empty) set of tuples, each containing a hole name $h$, a list of argument symbols $\vec{v}_h$, and a grammar $G_h$. A hole represents an uninterpreted function over the arguments $\vec{v}_h$. Each hole is associated with exactly one action $A_h$ and it appears exactly once in that action. The grammar of a hole defines the set of candidate expressions that can fill the hole. Because Holes is finite, we will assume without loss of generality that all holes are assigned an index to order them: $h_1, ..., h_n$.

For example, a sketch can be derived from Fig. 1 by replacing the update of line 8 with $vote\_yes' = h(vote\_yes, n)$, where $h$ is the hole name, the hole has arguments *vote_yes* and $n$, and the action of the hole is *VoteYes*$(n)$. One possible grammar for this hole is (in Backus Normal Form):

$$E ::= \emptyset \mid \{n\} \mid vote\_yes \mid (E \cup E) \mid (E \cap E) \mid (E \setminus E)$$

which generates all standard set expressions over the empty set, the singleton set $\{n\}$, and the set *vote_yes*. We note that, in general, each hole of a sketch may have its own distinct grammar.

A hole is either a *pre-hole* or a *post-hole*. If the hole is a pre-hole, it is a placeholder for a pre-condition of the action. If the hole is a post-hole, it is a placeholder for the right-hand side of a post-clause of the action, e.g., as in $vote\_yes' = h(vote\_yes, n)$, where $h$ is a post-hole. We write $h.var$ to denote the variable associated with a post-hole $h$. For instance, $h.var = vote\_yes$ for the post-hole $h$ shown prior. We do not consider synthesis of the initial state predicate and therefore no holes appear in Init. The arguments of a hole $h$ may include any of the the state variables in Vars. If $h$ is a pre-hole, then it returns a boolean. If the hole is a post-hole, its type is the same as its associated variable, e.g., hole $h$ above has the same type as *vote_yes*.

A *completion* of a sketch is a protocol derived from the sketch by replacing each hole with an expression from its grammar. If the sketch is clear from context or irrelevant, we may write the completion as a tuple $(e_1, \ldots, e_n)$, where $e_i$ is the expression filling the $i$th hole. Informally, the synthesis task is to find a completion of the protocol that satisfies a given property.

*Problem 1.* Let $\langle \textsc{Vars}, \textsc{Holes}, \textsc{Init}, \textsc{Next}_0 \rangle$ be a sketch and $\Phi$ a property. If one exists, find a completion of the sketch that satisfies $\Phi$. Otherwise, recognize that all completions violate $\Phi$.

We briefly note that Problem 1 asks to synthesize protocols that are only guaranteed to be correct for a particular choice of parameters (e.g. the number of nodes in the protocol). [12] takes a similar approach: (1) solve Problem 1 and then (2) use the TLA$^+$ Proof System (TLAPS) to show that the synthesized protocol is correct for all parameter choices. Our work in this paper improves step (1) of this approach, and we refer the reader to [12] for details on step (2). So, when we say "completion satisfies $\Phi$," we only guarantee that the completion satisfies $\Phi$ for the parameter choice used in the synthesis process.

## 3 Our Approach

The high-level architecture of our approach is shown in Fig. 2. Our method is a counterexample-guided inductive synthesis (CEGIS) algorithm, so it coordinates between a learner and verifier. We use the TLC model checker as our verifier [46]. Our learner works by coordinating between (1) an expression enumerator, (2) a counterexample generalizer, and (3) a search space reduction maintainer. The expression enumerator essentially generates candidate protocols by exploring the non-terminals of the protocol sketch's grammars in a breadth-first manner. The search space reduction is used by the expression enumerator to explore the grammars in a way that avoids generating redundant expressions. The counterexample generalizer converts counterexamples to logical constraints on expressions. The expression enumerator uses these *pruning constraints* along with a *constraint*

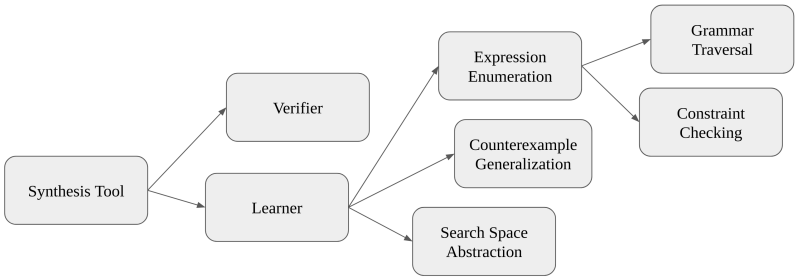


**Fig. 2.** The architecture of our method.

```
1  SYNTH(S, Φ) :=                          6      r, correct := CHECK(e, Φ)
2    U := init_search_space(S)             7      if correct then return e
3    while True:                           8      π := GENERALIZE(r, e, S)
4      e := PICK(U)                        9      U := PRUNE(U, π)
5      if e = ⊥ then return ⊥              10     U := ABSTRACT(U, π)
```

**Fig. 3.** Our algorithm.

*checker* to eliminate candidate protocols that exhibit previously encountered counterexamples, before they make it to the verifier.

Fig. 3 provides procedural details of our approach. The top-level procedure SYNTH takes as input a sketch $S$ and a property $\Phi$. The goal is to find a completion of $S$ that satisfies $\Phi$. In a nutshell, SYNTH works by generating candidate expressions $e$ from the hole grammars (line 4) until either the search space $U$ becomes empty, which means there exists no correct completion (line 5), or until a correct completion is found (line 7). If the current $e$ is incorrect, the counterexample $r$ returned by the verifier (line 6) is *generalized* into a *pruning constraint* $\pi$ which helps prune as well as reduce the search space (lines 9-10).

The PICK procedure generates expressions that (1) do not exhibit previously encountered counterexamples and (2) are distinct up to interpretation equivalence. Objective (1) is achieved by maintaining a set of *pruning constraints*, while Objective (2) is achieved by maintaining an *equivalence reduction*. The PICK procedure is able to check if a candidate expression satisfies the accumulated pruning constraints, but the GENERALIZE procedure is responsible for computing the pruning constraints and the PRUNE procedure for applying them to the search space. Likewise, PICK is able to explore the equivalence classes of expressions induced by the interpretation reduction, but the ABSTRACT procedure is responsible for actually maintaining the reduction.

In Section 4 we describe pruning constraints and the GENERALIZE routine in more detail. Then in Section 5 we describe the representation of the interpretation reduction, how we explore the reduced search space, and how the algorithm maintains the reduction as it encounters new counterexamples. Finally, we discuss the soundness, completeness, and termination of our algorithm in Section 6.

## 4 Pruning Constraints

**Counterexamples** A counterexample is a finite run $r$ annotated with actions: $s_0 \xrightarrow{A_1} s_1 \ldots \xrightarrow{A_k} s_k$. We support four types of counterexamples: safety, deadlock, liveness, and stuttering. $r$ is a *safety violation* of protocol $e$ if $r$ is a run of $e$ and $r$ violates a safety property. $r$ is a *deadlock violation* of protocol $e$ if $r$ is a run of $e$ and no action is enabled in $s_k$. $r$ is a *liveness violation* of protocol $e$ if (1) $r$ is a run of $e$, (2) a *loop* of $r$ is an *unfair cycle*, and (3) the infinite run induced by the cycle violates a liveness property. A loop of $r$ is a suffix of $r$ that starts and ends at the same state. A loop is a *strongly fair cycle* if there exists a strongly fair action $A$ that is not taken in the loop, but is enabled at some state in the loop. A loop is a *weakly fair cycle* if there exists a weakly fair action $A$ that is not taken in the loop, but is enabled at all states in the loop. A loop is an unfair cycle if it neither a strongly, nor weakly fair cycle. A run $r$ is a *stuttering violation* of protocol $e$ if (1) $r$ is a run of $e$, (2) all actions that are fair and enabled in $s_k$ transition back to $s_k$ (they are self-loops), (3) the infinite run induced by a self-loop at $s_k$ violates a liveness property, and (4) $s_k$ is not a deadlock state.

**Pruning Constraints** The syntax of pruning constraints is as follows, where $h$ is any hole name, $\alpha$ is an assignment of values to the arguments of $h$, and $c$ is a constant value: $\pi ::= d \mid \pi \vee \pi \mid \pi \wedge \pi$ and $d ::= h(\alpha) \neq c$. A completion $(e_1, ..., e_n)$ satisfies a pruning constraint of the form $\pi_1 \vee \pi_2$ if and only if it satisfies $\pi_1$ or $\pi_2$. Likewise for $\pi_1 \wedge \pi_2$, but satisfying both $\pi_1$ and $\pi_2$. The completion satisfies a pruning constraint of the form $h_i(\alpha) \neq c$ if and only if $e_i(\alpha) \neq c$, where $e(\alpha)$ denotes the result of substituting the values of $\alpha$ into $e$.

GENERALIZE$(r, e, S)$ constructs a pruning constraint as a function of the counterexample $r$, the completion $e$, and the sketch $S$. We assume that the expression in $e$ for hole $h$ is $e_h$. We also assume that the pre- and post-holes of $S$ are $pre(S)$ and $post(S)$ respectively. Finally, we assume access to the expressions for the fixed, non-hole pre- and post-conditions of the actions in $S$.

At a high level, our pruning constraints encode (1) which pre-conditions should be disabled, (2) which pre-conditions should be enabled, and (3) which post-conditions should evaluate to a different value in future completions. Our pruning constraints are each a disjunction of clauses that specify these conditions and hence we use the following three parametric clauses as *gadgets*:

$$\chi_{move}(Acts, \Xi, e) := \bigvee_{A \in Acts} \bigvee_{s \in \Xi} \bigvee_{h \in post(A)} h(s) \neq e_h(s)$$

$$\chi_{disable}(Acts, \Xi) := \bigvee_{A \in Acts} \bigvee_{s \in \Xi} \bigvee_{h \in pre(A)} h(s) \neq True$$

$$\chi_{enable}(Acts, \Xi) := \bigvee_{A \in Acts} \bigwedge_{s \in \Xi} \bigwedge_{h \in pre(A)} h(s) \neq False$$

Intuitively, the gadget $\chi_{move}$ says that some action $A$ directs a predecessor state $s$ to some successor state $s'$ that is different from the one specified by the completion $e$. Formally, $\chi_{move}(Acts, \Xi, e)$ says there must exist an action $A$ in $Acts$,

a state $s$ in $\Xi$, and a post-hole $h$ in $A$ such that $h(s) \neq e_h(s)$. Note: a state $s$ is an assignment of state variables to values, $e_h$ is the expression chosen for hole $h$, and $e_h(s)$ is the result of substituting $s$ into $e_h$.

The gadget $\chi_{disable}(Acts, \Xi)$ says there must exist an action $A$ in $Acts$, a state $s$ in $\Xi$, and a pre-hole $h$ in $A$ such that $h$ does not evaluate to *True* at state $s$—i.e. $A$ is disabled at $s$ by virtue of $h$. Finally, $\chi_{enable}(Acts, \Xi)$ says that there must exist an action $A$ in $Acts$ such that for all states $s$ in $\Xi$ and all pre-holes $h$ in $A$, $h$ does not evaluate to *False* at $s$—i.e. $A$ is enabled at all states in $\Xi$ if non-hole pre-conditions allow.

**Generalization of Safety Violations** Then for a safety counterexample $r = s_0 \xrightarrow{A_1} s_1 \ldots \xrightarrow{A_k} s_k$ of completion $e$ of sketch $S$, we construct the pruning constraint $\pi_{safe}(r, e, S)$ as

$$\pi_{safe} := \bigvee_{i=0}^{k-1} \chi_{move}(\{A_{i+1}\}, \{s_i\}, e) \vee \chi_{disable}(\{A_{i+1}\}, \{s_i\})$$

which says that we need to move a transition to a different state or disable a transition in order to avoid the safety violation $r$.

**Generalization of Deadlock Violations** For a deadlock counterexample, we construct the pruning constraint $\pi_{dead}(r, e, S)$ as

$$\pi_{dead} := \pi_{safe} \vee \chi_{enable}(Acts_{dead}, \{s_k\})$$

where $Acts_{dead}$ is the set of *can-enable* actions in sketch $S$ at state $s_k$. An action $A$ is can-enable at $s_k$ if all (non-hole) pre-conditions of $A$ evaluate to *True* at $s_k$. This constraint says that we either need to disable the path to the deadlocked state or we need to enable an action that undeadlocks $s_k$.

**Generalization of Liveness Violations** For a liveness counterexample, we construct the pruning constraint $\pi_{live}(r, e, S)$ as

$$\pi_{live} := \pi_{safe} \vee \chi_{enable}(Acts_{weak}, cycle(r)) \vee \bigvee_{s \in cycle(r)} \chi_{enable}(Acts^{(s)}_{strong}, \{s\})$$

where $cycle(r)$ is the set of states in the loop of $r$, and $Acts^{(s)}_{strong}$ is the set of strongly fair, disabled, can-enable actions in $S$ at state $s$. This constraint says that we need to either (1) alter the lasso $r$, (2) enable a strongly fair action in some state of the loop, or (3) enable a weakly fair action in all states of the loop.

**Generalization of Stuttering Violations** Finally, for a stuttering counterexample, we construct the pruning constraint $\pi_{stut}(r, e, S)$ as

$$\pi_{stut} := \pi_{safe} \vee \chi_{move}(Acts_{enabled}, \{s_k\}, e) \vee \chi_{enable}(Acts_{disabled}, \{s_k\})$$

where $Acts_{enabled}$ is the set of enabled fair actions in state $s_k$ and $Acts_{disabled}$ is the set of (strongly or weakly) fair, disabled, can-enable actions in state $s_k$. Because $s_k$ is a stuttering state, we know that all enabled fair actions are stuttering on $s_k$ and that we can avoid the stuttering violation by having one of these actions move to a state other than $s_k$. We can also avoid the stuttering violation by enabling a fair action that is disabled at $s_k$.

Finally, we define GENERALIZE$(r, e, S)$ to be $\pi_{vtype}(r, e, S)$ where $vtype \in \{safe, dead, live, stut\}$ is the violation type of $r$.

**Lemma 1.** *Let $r$ be a safety counterexample of completion $e_1$ of sketch $S$. Let $\pi = \pi_{safe}(r, e_1, S)$. Then for all completions $e_2$ of $S$, $e_2$ satisfies $\pi$ if and only if $r$ is not a safety counterexample of $e_2$.*

**Lemma 2.** *Let $r$ be a deadlock counterexample of completion $e_1$ of sketch $S$. Let $\pi = \pi_{dead}(r, e_1, S)$. Then for all completions $e_2$ of $S$, $e_2$ satisfies $\pi$ if and only if $r$ is not a deadlock counterexample of $e_2$.*

**Lemma 3.** *Let $r$ be a liveness counterexample of completion $e_1$ of sketch $S$. Let $\pi = \pi_{live}(r, e_1, S)$. Then for all completions $e_2$ of $S$, $e_2$ satisfies $\pi$ if and only if $r$ is not a liveness counterexample of $e_2$.*

**Lemma 4.** *Let $\Phi$ be a property. Let $r$ be a stuttering counterexample of completion $e_1$ of sketch $S$. Let $\pi = \pi_{stut}(r, e_1, S)$. Then $e_1$ does not satisfy $\pi$. Furthermore, for all completions $e_2$ of $S$, if $e_2$ does not satisfy $\pi$, then $e_2$ does not satisfy $\Phi$.*

**Theorem 1.** *Let $r$ be a counterexample of completion $e_1$ of sketch $S$. Let $\pi =$ GENERALIZE$(r, e_1, S)$. Then $e_1$ does not satisfy $\pi$. Furthermore, if $r$ is not a stuttering counterexample, then for all completions $e_2$ of $S$, if $e_2$ satisfies $\pi$, then $r$ is not a counterexample of $e_2$.*

*Proof.* This follows from Lemmas 1-4. See [14] for proofs. □

**Theorem 2.** *Let $\Phi$ be a property. Let $r$ be a counterexample to $\Phi$ of completion $e_1$ of sketch $S$. Let $\pi =$ GENERALIZE$(r, e_1, S)$. Then for all completions $e_2$ of $S$, if $e_2$ does not satisfy $\pi$, then $e_2$ does not satisfy $\Phi$. Futhermore, if $r$ is not a stuttering counterexample and $e_2$ does not satisfy $\pi$, then $r$ is a counterexample of $e_2$.*

*Proof.* This follows from Lemmas 1-4. See [14] for proofs. □

**Exactness of Pruning Constraints** A pruning constraint $\pi$ is *under-pruning* with respect to a sketch $S$ and a counterexample $r$ if there exists a completion $e$ of $S$ such that $e$ satisfies $\pi$ but $r$ is a counterexample of $e$. A pruning constraint $\pi$ is *over-pruning* with respect to a sketch $S$ and a counterexample $r$ if there exists a completion $e$ of $S$ such that $e$ does not satisfy $\pi$ but $r$ is not a counterexample of $e$. A pruning constraint $\pi$ is *exact* (called "optimal" in [12]) if it is neither under-pruning nor over-pruning. The safety, deadlock, and liveness pruning constraints

$\pi_{safe}$, $\pi_{dead}$, and $\pi_{live}$, presented above, are all exact. In contrast, only the safety pruning constraints in prior work [12] are exact. The stuttering constraints $\pi_{stut}$ presented here are not exact, but they are sufficient for the correctness of our synthesis algorithm (see Section 6). In the full version of this paper [14], we provide an alternative to $\pi_{stut}$ that is exact. However, this alternative constraint introduces additional performance overhead without providing sufficient benefit. Therefore, we do not consider the alternative pruning constraint further here.

## 5 Interpretation Reduction

In this section we discuss how we represent, use, and maintain the reduced search space. We conclude the section by showing that our algorithm does not exclude any correct completions, which is always a risk of reduction techniques. In Section 6 we show that our algorithm is sound, complete, and terminating.

**Representation of the Reduced Search Space** For simplicity, we will temporarily assume that the sketch $S$ has a single hole $h$. In general, we maintain a reduced search space for each hole in the sketch. We represent the (reduced) search space $U$ (Fig. 3) as a tuple $\langle G, \mathcal{A}, V, \Pi \rangle$, where $G$ is the grammar of the hole, $\mathcal{A}$ is a list of interpretations, $V$ is a partial map from *annotated non-terminals* of $G$ to expressions, and $\Pi$ is the conjunction of all accumulated pruning constraints.

We write $e_1 \equiv_\alpha e_2$ if $e_1$ and $e_2$ are interpretation equivalent. If $\mathcal{A}$ is a set of interpretations, then $e_1$ and $e_2$ are interpretation equivalent with respect to $\mathcal{A}$ if for all $\alpha \in \mathcal{A}$, $e_1 \equiv_\alpha e_2$.

Intuitively, annotated non-terminals "label" the equivalence classes of expressions induced by interpretation equivalence under $\mathcal{A}$. Formally, an annotated non-terminal is a pair $\langle q, \vec{c} \rangle$, where $q$ is a non-terminal in the grammar $G$, and $\vec{c} = (c_1, \ldots, c_n)$ is a (possibly empty) tuple of constants. The number of constants $n$ is equal to the number of interpretations in $\mathcal{A}$. We write $[\![q, \vec{c}]\!]_{G,\mathcal{A}}$ to denote the equivalence class labeled by the annotated non-terminal $\langle q, \vec{c} \rangle$ and omit $G$ and $\mathcal{A}$ if they are clear from context.

If $\alpha_i$ is the $i$-th interpretation in $\mathcal{A}$ and $q$ is a non-terminal in the grammar $G$, we write $e \in [\![q, \vec{c}]\!]$ if and only if (1) for all $i$, $e$ evaluates to $c_i$ under $\alpha_i$ and (2) $e$ is generated by $q$ in $G$. The partial map $V$ has the following invariant: for every annotated non-terminal $\langle q, \vec{c} \rangle$ in the domain of $V$, the expression $e = V[\langle q, \vec{c} \rangle]$ is such that $e \in [\![q, \vec{c}]\!]$. Intuitively, $V$ keeps track of (1) which equivalence classes have been visited and (2) which enumerated expression represents that equivalence class.

**Enumerating Expressions** Recall that the PICK procedure is responsible for generating candidate completions (Fig. 3, line 4). PICK works by treating $V$ as a cache of enumerated expressions and using the rules of the hole grammar $G$ to enumerate larger expressions. For instance, if $G$ has a rule $q \to (q_1 + q_2)$, and

$V[\langle q_1, (1,0)\rangle] = x$ and $V[\langle q_2, (1,0)\rangle] = y$, then PICK will build the expression $x+y$ and set $V[\langle q, (2,0)\rangle] = x+y$. A critical detail is that we do not change $V$ if $\langle q, (2,0)\rangle$ is already in the domain of $V$. If an annotated non-terminal is already in $V$, the algorithm has already enumerated an expression that is interpretation equivalent to $x + y$ under $\mathcal{A}$ for the non-terminal $q$.

**Ensuring Completeness** Using interpretation reduction allows for improved performance, but we have to be careful to ensure that we are not erroneously excluding completions. If $\mathcal{A}$ does not contain the "appropriate" interpretations, then PICK may return $\perp$ prematurely. Consider a simple example of what can go wrong. Suppose $G$ is the grammar $E ::= x \mid E + 1$ and that we have the property $\Phi := e \neq 0$. $\Phi$ is violated by $e := x$, since $e$ evaluates to 0 under the interpretation $[x \mapsto 0]$. Now suppose $[x \mapsto 0]$ is not in $\mathcal{A}$, particularly suppose $\mathcal{A}$ is empty. Then PICK will set $V[\langle E, ()\rangle] = x$ and then try to use the rule $E \to E+1$ to generate $x+1$. However, $x+1$ and $x$ are interpretation equivalent under the empty set of interpretations, so PICK will return $\perp$ before enumerating $x + 1$. $x + 1$ satisfies $\Phi$, so the reduction has eliminated a valid completion.

We maintain the invariant that $\mathcal{A}$ contains all interpretations that appear in the pruning constraints. Specifically, $\mathit{interp}(\pi)$ is defined inductively on the structure of pruning constraints: $\mathit{interp}(h(\alpha) \neq c) = \{\alpha\}$, and $\mathit{interp}(\pi_1 \vee \pi_2) = \mathit{interp}(\pi_1 \wedge \pi_2) = \mathit{interp}(\pi_1) \cup \mathit{interp}(\pi_2)$. Without loss of generality, assume that $\mathit{interp}(\Pi)$ is an ordered list with no duplicates. The expressions in $\mathit{interp}(\Pi)$ are sufficient to ensure the reduction does not exclude any correct completions (see reasoning across Lemma 5, Lemma 7, and Theorem 4).

**Lemma 5.** *Suppose that $e_1 \equiv_{\mathcal{A}} e_2$ with $\mathcal{A} = \mathit{interp}(\Pi)$. Then $e_1 \vDash \Pi$ if and only if $e_2 \vDash \Pi$.*

We also maintain the invariant that if $\langle q, \vec{c}\rangle$ is in the domain of $V$, then $\vec{c}$ has the same length as $\mathcal{A}$. This invariant is maintained by extending $\vec{c}$ in all annotated non-terminals every time we add a new interpretation to $\mathcal{A}$. In particular, we extend $\vec{c}$ for each $\langle q, \vec{c}\rangle$ in the domain of $V$ by computing the value of $V[\langle q, \vec{c}\rangle]$ under any new interpretations.

**Expressions Modulo Interpretation Equivalence** A key contribution of our work is that our algorithm is provably complete and terminating, even when the input is an unrealizable synthesis problem. To prove these facts we first introduce some additional formalism.

We first define what it means for one set of expressions to subsume another set of expressions, modulo interpretation equivalence. Let $\mathcal{A}$ be a set of interpretations. Let $E_1$ and $E_2$ be two sets of expressions. We say that $E_1$ is a subset of $E_2$ modulo interpretation equivalence, written $E_1 \subseteq_{\mathcal{A}} E_2$, if for all $e_1 \in E_1$ there exists an $e_2 \in E_2$ such that $e_1 \equiv_{\mathcal{A}} e_2$.

Suppose that PICK returns $\perp$. We denote by $\mathcal{L}(V)$ the set of expressions contained in $V$ after termination and by $\mathcal{L}(G)$ the set of expressions generated by the grammar $G$. To guarantee completeness, we need to ensure that

we enumerate every expression in $\mathcal{L}(G)$ modulo interpretation equivalence—i.e., $\mathcal{L}(G) \subseteq_{\mathcal{A}} \mathcal{L}(V)$. Our algorithm for populating $V$ starts with the terminals of $G$ and works its way up to larger expressions, per standard grammar enumeration techniques. Therefore, if we omit the interpretation reduction, our algorithm will enumerate all expressions in $G$. The only time we might miss an expression is if we ignore it because it is interpretation equivalent to an expression we have already enumerated. Furthermore, if two expressions are interpretation equivalent, they are equally useful for generating new expressions that are distinct up to interpretation equivalence. Hence, $\mathcal{L}(G) \subseteq_{\mathcal{A}} \mathcal{L}(V)$. The following corollary follows immediately from this observation.

**Corollary 1.** *If $e_1 \in \mathcal{L}(G) \setminus \mathcal{L}(V)$, then there exists an $e_2 \in \mathcal{L}(G) \cap \mathcal{L}(V)$ such that $e_1$ is interpretation equivalent to $e_2$ under $\mathcal{A}$.*

**Lemma 6.** *Suppose that if $e \not\models \Pi$, then $e \not\models \Phi$. Also suppose that $\mathcal{A} = interp(\Pi)$. Finally, suppose that for all $e \in \mathcal{L}(G) \cap \mathcal{L}(V)$, $e \not\models \Pi$. Then for all $e \in \mathcal{L}(G)$, $e \not\models \Phi$.*

*Proof.* Suppose that $e_1 \in \mathcal{L}(G)$. Either $e_1 \in \mathcal{L}(V)$ or $e_1 \notin \mathcal{L}(V)$. If $e_1 \in \mathcal{L}(V)$, then $e_1 \not\models \Pi$ and hence $e_1 \not\models \Phi$. Otherwise, $e_1 \notin \mathcal{L}(V)$ and by Corollary 1: $e_1 \equiv_{\mathcal{A}} e_2$ for some $e_2 \in \mathcal{L}(G) \cap \mathcal{L}(V)$. By assumption, $e_2 \not\models \Pi$ and therefore by Lemma 5, $e \not\models \Pi$. So $e \not\models \Phi$. □

**Lemma 7.** *If $\textsc{Pick}(U) = \bot$, then there is no completion of the sketch $S$ that satisfies the property $\Phi$.*

*Proof.* This lemma follows from Theorem 2 and Lemma 6, which say, respectively, that neither the pruning constraints nor the interpretation reduction exclude any correct completions. □

## 6 Soundness, Completeness, and Termination

The soundness of our method is relatively straightforward.

**Theorem 3.** *If $\textsc{Synth}(S, \Phi) = e$ and $e \neq \bot$, then $e$ is a completion of the sketch $S$ and $e$ satisfies the property $\Phi$.*

*Proof.* This theorem follows from (1) expressions chosen by the Pick subroutine come from the hole grammars of the sketch and (2) every completion is model checked against $\Phi$ before being returned. □

The completeness of our method follows from the completeness of the Pick subroutine.

**Theorem 4.** *If $\textsc{Synth}(S, \Phi) = \bot$, then there is no completion of the sketch $S$ that satisfies the property $\Phi$.*

*Proof.* Synth returns $\bot$ only if the Pick subroutine returns $\bot$. So by Lemma 7, there is no completion of the sketch $S$ that satisfies the property $\Phi$. □

Finally, under the assumptions specified in the following theorem, our method is guaranteed to terminate.

**Theorem 5.** *Suppose that for each set of interpretations $\mathcal{A}$, there are only finitely many distinct expressions in $G$, up to interpretation equivalence. Then the* Synth *algorithm terminates for any input sketch $S$ and property $\Phi$.*

*Proof.* We synthesize protocol instances with finite domain state variables, so the size of $\mathcal{A}$ is bounded. So eventually the $\mathcal{A}$ will be updated for the last time. There are only finitely many distinct expressions in $G$, up to interpretation equivalence under the final $\mathcal{A}$. Therefore, the Pick subroutine will eventually return a correct completion or $\perp$. □

We remark that Theorem 5 does not contradict the undecidability of Problem 1. We guarantee termination only for sketches with finitely many distinct expressions, up to interpretation equivalence. All the benchmarks used in our experiments satisfy this condition.

## 7 Evaluation

We implement our method in a tool called PolySemist. In this section, we compare PolySemist to Scythe, a state-of-the-art distributed protocol synthesis tool [12]. Scythe is publicly available [11]. We evaluate the performance of the two tools on a set of benchmarks taken from [12]. Our experiments come from seven benchmark protocols: two phase commit (2PC), consensus, simple decentralized lock (DL), lock server (LS), sharded key value store (SKV), and two reconfigurable raft protocols (RR and RR-big). For RR and RR-big we use the same incomplete protocols as in [12]. Otherwise, the easiest benchmarks have all pre- or all post-conditions missing from one action of the protocol. The hardest benchmarks have all pre- and post-conditions missing from two actions. PolySemist and Scythe are both written in Python and all experiments are run on a dedicated 2.4GHz CPU.

We conduct two types of experiments: (realizable) synthesis experiments, and unrealizability experiments. In synthesis experiments, we compare the execution time of PolySemist and Scythe on realizable synthesis problems. PolySemist is faster than Scythe in 160 out of 171 realizable synthesis experiments. Of the 11 experiments where Scythe is faster, the difference in runtimes is more than 10 seconds in just 2 cases.

In the unrealizability experiments, we compare the ability of PolySemist and Scythe to detect unrealizability. Our tool was able to detect unrealizability in 80 out of 123 instances and timed out after 1 hour in the remaining 43 instances. Scythe recognized unrealizability in 16 of the 123 instances and timed out after 1 hour in the remaining 107 instances.

### 7.1 Realizable Synthesis Experiments

We partition our synthesis experiments into three categories: short run experiments (Fig. 4) with a total execution time of less than two minutes, longer

experiments where both tools found a solution (Fig. 5), and still heavier experiments (Fig. 6), in which Scythe always times out. In these figures, number labels on the horizontal axis correspond to distinct synthesis problems and every other tick is skipped in Fig. 4 for readability.

There are 171 pairs of bars shown across the three figures. Each pair corresponds to an input to the synthesis tools (a distinct sketch-property pair). The left bar (gray, dotted) in each pair shows the execution time of Scythe, and the right bar (black, solid) shows the execution time of PolySemist. We ran each of the experiments five times and report the minimum time for both tools, as if we ran each tool five times in parallel and halted after the first run halted. In all cases where PolySemist timed out in all five runs, Scythe also did.

PolySemist performed better than Scythe in 160 out of 171 experiments. In Fig. 5, indices 11 and 23 show the 2 cases where Scythe is faster than PolySemist by more than 10 seconds. These two experiments where Scythe does much better use a variation of the consensus protocol sketch. Even so, there are several experiments where PolySemist is faster than Scythe on the consensus protocol by more than an order of magnitude (e.g., in Fig. 5 at index 21, PolySemist is 67 times faster than Scythe).

Across all experiments in Figs. 4 and 5 (cases where both tools succeeded), the total execution time for Scythe and PolySemist are about 32,000 and 9,000 seconds respectively. PolySemist is routinely faster by a factor of 100 or more.

Scythe times out in all five runs of all experiments of Fig. 6, hence Fig. 6 only includes bars for PolySemist. In the first 6 experiments shown in Fig. 6, PolySemist finished in less than 36 seconds, while Scythe timed out after 1 hour. In Fig. 6 at index 1, PolySemist finished in 18 seconds, while Scythe timed out after 1 hour; a speedup of at least 200 times.

The experiment at index 36 of Fig. 6 is to synthesize the entire distributed lock (DL) protocol *from scratch*. DL contains 2 pre-conditions and 4 post-conditions across 2 actions, so a total of 6 expressions need to be synthesized. Scythe times out after 1 hour, but PolySemist synthesizes the entire protocol in 125 seconds ($< 3$ minutes). To our knowledge, ours is the first tool to synthesize an entire TLA$^+$ protocol from scratch.

### 7.2 Unrealizability Experiments

There are several reasons a synthesis problem might be unrealizable and these reasons can be divided into two broad categories. In the first category, the chosen sketch grammar is not expressive enough, but there exists a grammar that would make the synthesis problem realizable. In the second category, the fixed parts of the sketch surrounding the holes are such that *no sketch grammar* is expressive enough.

The second category can be further divided: (1) there aren't enough state variables, (2) there aren't enough actions, (3) the fixed pre- and post-conditions are simply wrong, e.g., a pre-condition always evaluates to false, (4) a parameterized action is missing an argument, etc.

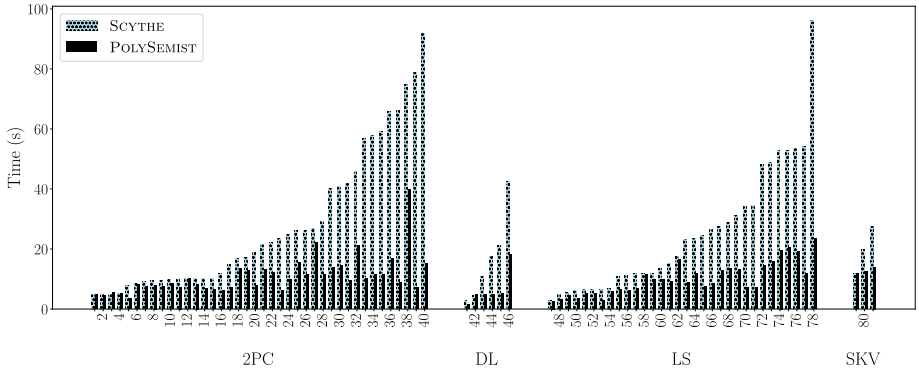


**Fig. 4.** Execution time comparison for short run experiments (total execution time less than 120 seconds). Only even ticks shown on horizontal axis.

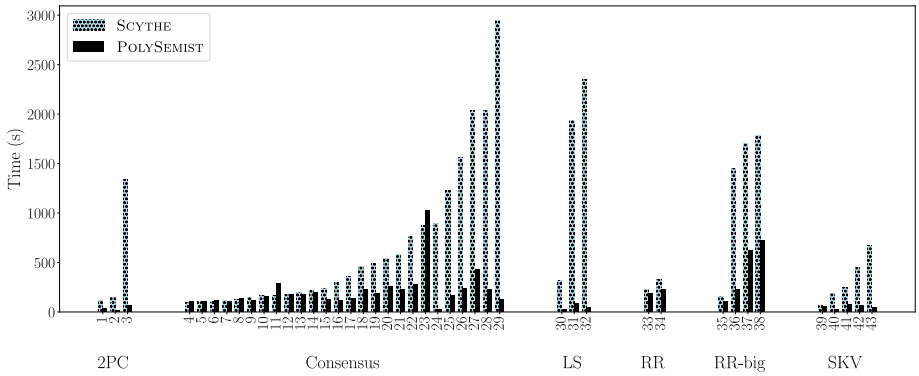


**Fig. 5.** Execution time comparison for longer run experiments where neither tool timed out (1 hour) in all five runs.

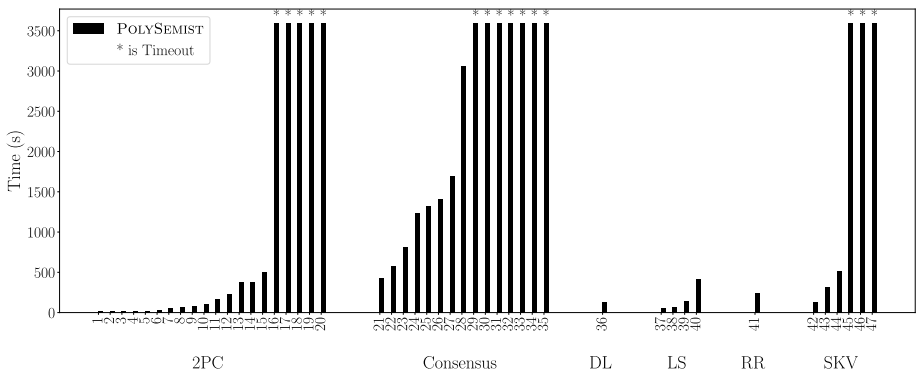


**Fig. 6.** Execution times for PolySemist in experiments where all five runs of Scythe timed out after 1 hour. Scythe always times out here, so its bars are not shown.

We construct unrealizable synthesis problems across these categories by transforming our realizable synthesis problems. These transformations include: (1) removing rules from the sketch grammar, (2) removing state variables from the sketch, (3) removing actions from the sketch, (4) changing the fixed pre- and post-conditions, and (5) removing parameters from actions. All of these transformations reflect omissions that a reasonable user might make when constructing a sketch.

In total, we ran 171 realizable synthesis problems in the previous section. We adapted these to obtain 123 unrealizable synthesis problems. There isn't a one-to-one correspondence between realizable and unrealizable synthesis problems because some transformations remove actions and these actions have holes.

PolySemist was able to detect unrealizability in 80 out of 123 instances and timed out after 1 hour in the remaining 43 instances. The experiments are partitioned by easy, medium, and hard benchmarks, which were derived from the experiments in Figs. 4, 5, and 6 respectively. PolySemist succeeded in 46/50 easy experiments, 21/35 medium experiments, and 13/38 hard experiments. In contrast, Scythe recognized unrealizability in 0/50, 9/35, and 7/38 easy, medium, and hard experiments respectively. Scythe timed out after 1 hour in the remaining 107 instances.

PolySemist terminated in all cases where Scythe did. In these cases where both terminated, the execution times of the tools were within 1 second of each other and they both terminated in less than 10 seconds. I.e., Scythe only terminated in cases where the unrealizability was particularly easy to detect.

## 8 Related Work

[2,5,13,16,17] study synthesis of explicit-state machines. TRANSIT [42] requires a human in the loop to handle counterexamples. All of [30,6,24,29] consider special classes of distributed protocols. [4] uses an ad-hoc specification language and relies on an external SyGuS solver to translate explicit input-output tables representing expressions into symbolic expressions. Works like [26] and [23] use methods that are not guaranteed to find a solution even if one exists.

Unrealizability results are not typically reported in the distributed protocol synthesis literature. For instance, all results reported in [2,5,42,30,6,24,4,26,23] are for realizable instances. Some related work reports results for unrealizable instances, but the target systems are either explicit-state [13,16,17] or from a special class of distributed protocols [29].

Prior to our work, [12] is the only work that synthesizes general purpose, symbolic distributed protocols written in TLA$^+$. Compared to that of [12], our approach has several major differences. First, our method uses interpretation equivalence, while [12] uses universal equivalence. Second, our counterexample generalization is exact except for stuttering counterexamples whereas that of [12] is only exact for safety counterexamples (c.f. Theorems 1 & 2, and related discussion). Third, our procedure is guaranteed to terminate under given conditions (satisfied in all our benchmarks), whereas under these same conditions,

the method of [12] may not terminate (and indeed times out in many experiments). Finally, our experimental results show that our approach is empirically better than that of [12] both in realizable and unrealizable problem instances.

Existing SyGuS solvers use SMT formulas to express properties, and are therefore not directly applicable to distributed protocol synthesis which requires temporal logic properties. But our techniques for generating expressions and checking them against pruning constraints are generally related to term enumeration strategies used in SyGuS [1]. Both EUSolver [3] and cvc4sy [35] are SyGuS solvers that generate larger expressions from smaller expressions. EUSolver uses divide-and-conquer techniques in combination with decision tree learning and is quite different from our approach. To our knowledge, EUSolver does not employ equivalence reduction at all. The "fast term enumeration strategy" of cvc4sy is similar to our cache-like treatment of $V$ and also uses an equivalence reduction technique. [21,22,27,31] can recognize unrealizable SyGuS problems, but do not handle temporal logics required for distributed protocol synthesis. To our knowledge, none of these approaches use anything like interpretation equivalence to reduce the search space. Absynthe [15] uses a fixed abstraction provided by the user to guide synthesis of programs from source languages with complex semantics (e.g., Python), albeit not for distributed protocols. Our abstraction (the interpretation reduction) is automatically generated and always changing based on the accumulated counterexamples.

## 9 Conclusion

We presented a novel CEGIS-based synthesis method for distributed protocols. We demonstrated that our method is able to synthesize protocols faster than the state of the art: in some cases, by several orders of magnitude. In one case we were able to synthesize an entire TLA$^+$ protocol from scratch in less than 3 minutes where the state of the art timed out after an hour.

We also provided conditions, satisfied by our benchmarks, under which our method is guaranteed to terminate, even in cases where the synthesis problem has no solution; the state of the art is not guaranteed to terminate under these conditions and makes no guarantees about termination in general. In practice, our method recognizes five times more unrealizable synthesis instances than the state of the art.

Our results are enabled first and foremost by a novel search space reduction technique called interpretation reduction; we proved that this technique does not compromise the completeness of the synthesis algorithm. Additionally, we use an advanced method for generalizing counterexamples.

For future work, we plan to investigate more sophisticated techniques for traversing the reduced search space. We are also investigating how to synthesize protocols when the actions and state variables are not known in advance.

**Acknowledgments.** This material is partly supported by the National Science Foundation under Graduate Research Fellowship Grant #1938052, and Award #2319500. Any opinion, findings, and conclusions or recommendations expressed in this material are those of the authors(s) and do not necessarily reflect the views of the National Science Foundation.

## References

1. Alur, R., Bodík, R., Juniwal, G., Martin, M.M.K., Raghothaman, M., Seshia, S.A., Singh, R., Solar-Lezama, A., Torlak, E., Udupa, A.: Syntax-guided synthesis. In: Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013. pp. 1–8. IEEE (2013), https://ieeexplore.ieee.org/document/6679385/
2. Alur, R., Martin, M., Raghothaman, M., Stergiou, C., Tripakis, S., Udupa, A.: Synthesizing Finite-state Protocols from Scenarios and Requirements. In: Haifa Verification Conference. LNCS, vol. 8855. Springer (2014)
3. Alur, R., Radhakrishna, A., Udupa, A.: Scaling enumerative program synthesis via divide and conquer. In: Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017. Lecture Notes in Computer Science, vol. 10205, pp. 319–336 (2017). https://doi.org/10.1007/978-3-662-54577-5_18, https://doi.org/10.1007/978-3-662-54577-5_18
4. Alur, R., Raghothaman, M., Stergiou, C., Tripakis, S., Udupa, A.: Automatic completion of distributed protocols with symmetry. In: Kroening, D., Pasareanu, C.S. (eds.) Computer Aided Verification - 27th International Conference, CAV. Lecture Notes in Computer Science, vol. 9207, pp. 395–412. Springer (2015). https://doi.org/10.1007/978-3-319-21668-3_23, https://doi.org/10.1007/978-3-319-21668-3_23
5. Alur, R., Tripakis, S.: Automatic synthesis of distributed protocols. SIGACT News **48**(1), 55–90 (2017). https://doi.org/10.1145/3061640.3061652, https://doi.org/10.1145/3061640.3061652
6. Bloem, R., Braud-Santoni, N., Jacobs, S.: Synthesis of self-stabilising and byzantine-resilient distributed systems. In: Chaudhuri, S., Farzan, A. (eds.) Computer Aided Verification - 28th International Conference, CAV. Lecture Notes in Computer Science, vol. 9779, pp. 157–176. Springer (2016). https://doi.org/10.1007/978-3-319-41528-4_9, https://doi.org/10.1007/978-3-319-41528-4_9
7. Buchman, E.: Tendermint: Byzantine fault tolerance in the age of blockchains (2016), https://api.semanticscholar.org/CorpusID:59082906
8. Buterin, V.: Ethereum white paper: A next generation smart contract & decentralized application platform (2013), https://github.com/ethereum/wiki/wiki/White-Paper
9. Corbett, J.C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J.J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P., et al.: Spanner: Google's globally distributed database. ACM Transactions on Computer Systems (TOCS) **31**(3), 1–22 (2013)
10. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: Amazon's highly available key-value store. ACM SIGOPS operating systems review **41**(6), 205–220 (2007)

11. Egolf, D.: scythe-fmcad2024. https://github.com/egolf-cs/scythe-fmcad2024
12. Egolf, D., Schultz, W., Tripakis, S.: Efficient synthesis of symbolic distributed protocols by sketching. In: Narodytska, N., Rümmer, P. (eds.) Proceedings of the 24th Conference on Formal Methods in Computer-Aided Design – FMCAD 2024. pp. 281–291. TU Wien Academic Press (2024). https://doi.org/10.34727/2024/isbn.978-3-85448-065-5_34, https://doi.org/10.34727/2024/isbn.978-3-85448-065-5_34
13. Egolf, D., Tripakis, S.: Synthesis of distributed protocols by enumeration modulo isomorphisms. In: ATVA 2023 - Part I. pp. 270–291. Lecture Notes in Computer Science, Springer (2023). https://doi.org/10.1007/978-3-031-45329-8_13, https://doi.org/10.1007/978-3-031-45329-8_13
14. Egolf, D., Tripakis, S.: Accelerating protocol synthesis and detecting unrealizability with interpretation reduction (2025), https://arxiv.org/abs/2501.14585
15. Fedchin, A., Dean, T., Foster, J.S., Mercer, E., Rakamaric, Z., Reger, G., Rungta, N., Salkeld, R., Wagner, L., Waldrip, C.: A toolkit for automated testing of dafny. In: Rozier, K.Y., Chaudhuri, S. (eds.) NASA Formal Methods - 15th International Symposium, NFM 2023, Houston, TX, USA, May 16-18, 2023, Proceedings. Lecture Notes in Computer Science, vol. 13903, pp. 397–413. Springer (2023). https://doi.org/10.1007/978-3-031-33170-1_24, https://doi.org/10.1007/978-3-031-33170-1_24
16. Finkbeiner, B., Schewe, S.: Bounded synthesis. Int. J. Softw. Tools Technol. Transf. **15**(5-6), 519–539 (2013). https://doi.org/10.1007/S10009-012-0228-Z, https://doi.org/10.1007/s10009-012-0228-z
17. Finkbeiner, B., Tentrup, L.: Detecting unrealizability of distributed fault-tolerant systems. Log. Methods Comput. Sci. **11**(3) (2015). https://doi.org/10.2168/LMCS-11(3:12)2015, https://doi.org/10.2168/LMCS-11(3:12)2015
18. Goel, A., Sakallah, K.: On Symmetry and Quantification: A New Approach to Verify Distributed Protocols. In: NASA Formal Methods: 13th International Symposium, NFM 2021. p. 131–150 (2021)
19. Gulwani, S., Polozov, O., Singh, R.: Program synthesis. Foundations and Trends in Programming Languages **4**(1-2), 1–119 (2017). https://doi.org/10.1561/2500000010
20. Hance, T., Heule, M., Martins, R., Parno, B.: Finding Invariants of Distributed Systems: It's a Small (Enough) World After All. In: 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21). pp. 115–131. USENIX Association (Apr 2021), https://www.usenix.org/conference/nsdi21/presentation/hance
21. Hu, Q., Breck, J., Cyphert, J., D'Antoni, L., Reps, T.W.: Proving unrealizability for syntax-guided synthesis. In: Dillig, I., Tasiran, S. (eds.) Computer Aided Verification - 31st International Conference, CAV. Lecture Notes in Computer Science, vol. 11561, pp. 335–352. Springer (2019). https://doi.org/10.1007/978-3-030-25540-4_18, https://doi.org/10.1007/978-3-030-25540-4_18
22. Hu, Q., Cyphert, J., D'Antoni, L., Reps, T.W.: Exact and approximate methods for proving unrealizability of syntax-guided synthesis problems. In: Donaldson, A.F., Torlak, E. (eds.) Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020. pp. 1128–1142. ACM (2020). https://doi.org/10.1145/3385412.3385979, https://doi.org/10.1145/3385412.3385979

23. Hui, Y., Ripberger, D., Lu, X., Wang, Y.: Learning distributed protocols with zero knowledge. In: Machine Learning for Systems at NeurIPS 2023 (2023), https://openreview.net/forum?id=u0Ncut8ru5
24. Jaber, N., Wagner, C., Jacobs, S., Kulkarni, M., Samanta, R.: Synthesis of distributed agreement-based systems with efficiently-decidable verification. In: TACAS 2023. Lecture Notes in Computer Science, vol. 13994, pp. 289–308. Springer (2023), https://doi.org/10.1007/978-3-031-30820-8_19
25. Jacobs, S., Bloem, R.: Parameterized synthesis. Log. Methods Comput. Sci. **10**(1) (2014). https://doi.org/10.2168/LMCS-10(1:12)2014, https://doi.org/10.2168/LMCS-10(1:12)2014
26. Katz, G., Peled, D.: Synthesizing solutions to the leader election problem using model checking and genetic programming. In: Haifa Verification Conference. p. 117–132. HVC'09, Springer (2009)
27. Kim, J., D'Antoni, L., Reps, T.W.: Unrealizability logic. Proc. ACM Program. Lang. **7**(POPL), 659–688 (2023). https://doi.org/10.1145/3571216, https://doi.org/10.1145/3571216
28. Lamport, L.: Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley (Jun 2002)
29. Lazic, M., Konnov, I., Widder, J., Bloem, R.: Synthesis of distributed algorithms with parameterized threshold guards. In: 21st International Conference on Principles of Distributed Systems, OPODIS. LIPIcs, vol. 95, pp. 32:1–32:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2017). https://doi.org/10.4230/LIPICS.OPODIS.2017.32, https://doi.org/10.4230/LIPIcs.OPODIS.2017.32
30. Mirzaie, N., Faghih, F., Jacobs, S., Bonakdarpour, B.: Parameterized synthesis of self-stabilizing protocols in symmetric networks. Acta Informatica **57**(1-2), 271–304 (2020). https://doi.org/10.1007/S00236-019-00361-7, https://doi.org/10.1007/s00236-019-00361-7
31. Nagy, S., Kim, J., D'Antoni, L., Reps, T.W.: Automating unrealizability logic: Hoare-style proof synthesis for infinite sets of programs. CoRR **abs/2401.13244** (2024). https://doi.org/10.48550/ARXIV.2401.13244, https://doi.org/10.48550/arXiv.2401.13244
32. Newcombe, C., Rath, T., Zhang, F., Munteanu, B., Brooker, M., Deardeuff, M.: How Amazon Web Services Uses Formal Methods. Commun. ACM **58**(4), 66–73 (Mar 2015). https://doi.org/10.1145/2699417, http://doi.acm.org/10.1145/2699417
33. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 179–190. POPL '89, Association for Computing Machinery, New York, NY, USA (1989). https://doi.org/10.1145/75277.75293, https://doi.org/10.1145/75277.75293
34. Pnueli, A., Rosner, R.: Distributed reactive systems are hard to synthesize. In: Proceedings of the 31th IEEE Symposium on Foundations of Computer Science. pp. 746–757 (1990)
35. Reynolds, A., Barbosa, H., Nötzli, A., Tinelli, C., Barrett, C.: CVC4SY: Smart and fast term enumeration for syntax-guided synthesis. In: Dillig, I., Tasiran, S. (eds.) Proceedings of the 31st International Conference on Computer Aided Verification (CAV). Lecture Notes in Computer Science, vol. 11561, pp. 74–83. Springer (Jul 2019). https://doi.org/10.1007/978-3-030-25543-5_5, http://theory.stanford.edu/~barrett/pubs/RBN+19.pdf

36. Schultz, W., Ashton, E., Howard, H., Tripakis, S.: Scalable, Interpretable Distributed Protocol Verification by Inductive Proof Slicing. arXiv eprint 2404.18048 (2024)
37. Schultz, W., Dardik, I., Tripakis, S.: Plain and Simple Inductive Invariant Inference for Distributed Protocols in TLA$^+$. In: 22nd Formal Methods in Computer-Aided Design, FMCAD 2022. pp. 273–283. IEEE (2022). https://doi.org/10.34727/2022/ISBN.978-3-85448-053-2_34, https://doi.org/10.34727/2022/isbn.978-3-85448-053-2_34
38. Solar-Lezama, A.: The sketching approach to program synthesis. In: Proceedings of the 7th Asian Symposium on Programming Languages and Systems. pp. 4–13. APLAS '09, Springer (2009)
39. Solar-Lezama, A.: Program sketching. Int. J. Softw. Tools Technol. Transf. **15**(5-6), 475–495 (oct 2013). https://doi.org/10.1007/s10009-012-0249-7, https://doi.org/10.1007/s10009-012-0249-7
40. Thistle, J.G.: Undecidability in decentralized supervision. Systems & Control Letters **54**(5), 503–509 (2005). https://doi.org/10.1016/j.sysconle.2004.10.002
41. Tripakis, S.: Undecidable Problems of Decentralized Observation and Control on Regular Languages. Information Processing Letters **90**(1), 21–28 (Apr 2004). https://doi.org/10.1016/j.ipl.2004.01.004
42. Udupa, A., Raghavan, A., Deshmukh, J.V., Mador-Haim, S., Martin, M.M.K., Alur, R.: TRANSIT: specifying protocols with concolic snippets. In: Boehm, H., Flanagan, C. (eds.) ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013. pp. 287–296. ACM (2013). https://doi.org/10.1145/2491956.2462174, https://doi.org/10.1145/2491956.2462174
43. Yao, J., Tao, R., Gu, R., Nieh, J.: DuoAI: Fast, Automated Inference of Inductive Invariants for Verifying Distributed Protocols. In: Aguilera, M.K., Weatherspoon, H. (eds.) 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2022). pp. 485–501. USENIX Association (2022), https://www.usenix.org/conference/osdi22/presentation/yao
44. Yao, J., Tao, R., Gu, R., Nieh, J.: Mostly automated verification of liveness properties for distributed protocols with ranking functions. Proceedings of the ACM on Programming Languages (POPL) **8**, 1028–1059 (jan 2024). https://doi.org/10.1145/3632877, https://doi.org/10.1145/3632877
45. Yao, J., Tao, R., Gu, R., Nieh, J., Jana, S., Ryan, G.: DistAI: Data-Driven Automated Invariant Learning for Distributed Protocols. In: 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2021). pp. 405–421. USENIX Association (Jul 2021), https://www.usenix.org/conference/osdi21/presentation/yao
46. Yu, Y., Manolios, P., Lamport, L.: Model Checking TLA+ Specifications. In: Pierre, L., Kropf, T. (eds.) Correct Hardware Design and Verification Methods. pp. 54–66. Springer Berlin Heidelberg, Berlin, Heidelberg (1999)