Efficient Synthesis of Symbolic Distributed Protocols by Sketching

Derek Egolf
Northeastern University
Boston, MA USA
egolf.d@northeastern.edu

William Schultz
Northeastern University
Boston, MA
schultz.w@northeastern.edu

Stavros Tripakis
Northeastern University
Boston, MA
stavros@northeastern.edu

***Abstract*—We present a novel and efficient method for synthesis of parameterized distributed protocols by sketching. Our method is both syntax-guided and counterexample-guided, and utilizes a fast equivalence reduction technique that enables efficient completion of protocol sketches, often significantly reducing the search space of candidate completions by several orders of magnitude. To our knowledge, our tool, SCYTHE, is the first synthesis tool for the widely used specification language TLA+. We evaluate SCYTHE on a diverse benchmark of distributed protocols, demonstrating the ability to synthesize a large scale distributed Raft-based dynamic reconfiguration protocol beyond the scale of what existing synthesis techniques can handle.**

***Index Terms*—distributed protocols, synthesis, syntax-guided, counterexample-guided, sketching**

I. INTRODUCTION

Distributed protocols have become a crucial component in the operation of modern computer systems, including financial infrastructure [9], [8] and cloud data storage systems [10], [12]. In addition to being consequential and widely used, the complexity of these protocols makes them notoriously hard to design and reason about.

Automated verification of distributed protocols has made great advances in recent years. Specifically, inductive invariant inference methods have allowed for fuller automation of the verification of safety properties [18], [21], [50], [48], [41], [39]. State of the art tools in this domain are able to verify non-trivial specifications of *parameterized*, *infinite-state* protocols, written in languages such as TLA$^+$ [28] or Ivy [35]. Such verification efforts include not just protocol specifications especially designed to fit into the decidable fragment of Ivy [34], but also generally undecidable specifications of protocols such as Raft written in TLA$^+$ [41], [39] or Paxos written in Ivy [34], [19]. Progress is also being made towards fuller automation of the verification of liveness properties, e.g., see [49].

On the other hand, automated *synthesis* of distributed protocols is less advanced. This discrepancy might be expected because synthesis is intuitively a harder problem than verification: verification is about checking that a *given* system is correct, while synthesis involves inventing a system *and* ensuring that it is correct. Theory supports this intuition: model checking finite-state distributed systems is decidable, but synthesis of finite-state distributed systems is generally undecidable [37], [45], [46]. Synthesis of parameterized distributed protocols is also generally undecidable [25]. But even when decidable, synthesis "from scratch" is still a harder problem than verification, e.g., single-module reactive synthesis from LTL specifications is *doubly* exponential in the size of the LTL formula [36].

An easier problem than doing synthesis from scratch is to do synthesis by sketching [42], [43]. Sketching turns the synthesis problem into a completion problem: given a *sketch* (i.e., an incomplete system with *holes*) the goal is to complete the sketch such that the completion satisfies a given correctness specification. The holes are typically missing state variable updates, guards, or parts thereof. Completing a hole means finding the missing expression.

In this paper, we consider the problem of synthesis of distributed protocols by sketching. Contrary to prior works that either apply only to special classes of protocols [30], [24], or target protocols in an ad-hoc specification language [4], our work targets general protocols written in TLA$^+$ [28], a highly expressive specification language with widespread use in both academia and the industry [32].

Our approach follows the counterexample-guided inductive synthesis (CEGIS) paradigm [43], [20]: a *learner* is responsible for generating candidate solutions, while a *verifier* is responsible for checking whether a candidate satisfies the requirements.

Our synthesis method is truly *syntax-guided* in the sense that our synthesis loop explores directly the space of candidate symbolic expressions that can be generated from a given grammar. In contrast, previous work [4] explores the space of (finite) interpretations of uninterpreted functions. Our synthesis tool generates expressions, whereas the synthesis tool of [4] generates input-output tables (which can then be passed to an external SyGuS solver [1] to obtain an expression as a post-processing step). Our method does not rely on an external SyGuS solver.

A crucial component of our synthesis algorithm is how exactly we generate candidate expressions from a given grammar (or grammars, in the case of multiple holes). A naive, breadth-first enumeration of all possible expressions in the grammar

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.

https://doi.org/10.34727/2024/isbn.978-3-85448-065-5_34

does not scale. Instead, we use a novel technique that employs *cache-based* enumeration coupled with an *equivalence reduction* with *short-circuiting*. This technique allows us to not only avoid checking semantically equivalent expressions, but also to avoid generating redundant expressions in the first place. Indeed, some of our experiments show a reduction in the number of generated expressions of more than three orders of magnitude.

We implement our method in a synthesis tool called SCYTHE. SCYTHE synthesizes protocols that are parameterized, i.e., in a form that is directly generalizable to an arbitrary number of nodes. Some of our synthesized protocols can be instantiated with infinite-domain variables, i.e., we can handle infinite-state protocols. SCYTHE is able to synthesize complex expressions from non-trivial grammars. For example, SCYTHE can synthesize the guard expression

$$\begin{aligned} &\forall Q_1 \in \mathit{Quorums}(\mathit{config}[i]), \\ &\quad \forall Q_2 \in \mathit{Quorums}(\mathit{new\_config}) : Q_1 \cap Q_2 \neq \emptyset \end{aligned} \tag{1}$$

which is parameterized by i, the node that is executing reconfiguration, and it can also synthesize the state variable update

$$\mathit{votes}' = [\mathit{votes} \text{ EXCEPT } [n_1] = \mathit{votes}[n_1] \cup \{n_2\}] \tag{2}$$

which is parameterized by n_1 and n_2, the nodes that are exchanging votes.

We evaluate SCYTHE on a suite of non-trivial benchmarks, including variants of the Raft dynamic reconfiguration protocol [33], [40], [41]. SCYTHE is able to synthesize correct and sometimes novel protocols in less than an hour and often in a matter of minutes. Although SCYTHE itself only guarantees correctness for a finite protocol instance, we were able to prove a-posteriori (using TLAPS [11]) that the synthesized protocols are in fact correct for an arbitrary number of nodes, as well as in some cases for infinite-domain state variables.

In summary, this work makes the following contributions: (1) A novel distributed protocol synthesis method that is both *syntax-guided* as well as *counterexample-guided*. (2) Novel techniques to accelerate the search of candidate completions, often reducing the search space by several orders of magnitude. (3) The synthesis tool SCYTHE which, to our knowledge, is the only tool able to handle a diverse suite of real-world distributed protocol benchmarks written in a broadly used language such as TLA$^+$. (4) Formal correctness proofs which demonstrate that SCYTHE is able to synthesize infinite-state, parameterized protocols that are safe for any protocol instance.

II. PRELIMINARIES

A. 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 $\langle$PARAMS, VARS, INIT, NEXT$\rangle$. PARAMS is a set of *parameters* that may vary from one instantiation of the protocol to the other, but do not change during the execution of the protocol (e.g. a set of node ids *Node* in Fig. 1 line 1). 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 Sections II-B and II-D.

TLA$^+$ is untyped, but for purposes of synthesis we assume that each symbol in PARAMS and VARS is typed. Supported types include *Bool* and *Int*, and sets or arrays of types. If *T1* and *T2* are types, then an element of type *(Set T1)* is a set of elements of type *T1* and an element of type *(Array T1 T2)* is a map from elements of type *T1* to elements of type *T2*.

A tuple $\langle$CONST, VARS, INIT, NEXT$\rangle$ denotes an *instance* of a protocol, where CONST is a mapping of PARAMS to values. For instance, $[\mathit{Node} \mapsto \{n1, n2, n3\}]$ characterizes one instance of the protocol in Fig. 1 and $[\mathit{Node} \mapsto \{a0, b0\}]$ characterizes another. The values of parameters need not be finite sets, e.g., *MaxVal* might have type *Int* and specify a bound on some value. Note that a protocol is technically not operational until the symbols in PARAMS are assigned to values, since the valuation of INIT and NEXT may depend on the valuation of the symbols in PARAMS.

A symbol in PARAMS may have type *Domain*, which designates it as an *opaque set*. If *Prm* is a PARAMS symbol of type *Domain* and CONST assigns *Prm* to set P, then an object x has type *(OfDomain Prm)* if and only if $x \in P$. For instance, if symbol *Node* in Fig. 1 has type *Domain*, then the symbol *vote_yes* has type *(Set (OfDomain Node))*. (Alternatively, *Node* could have type *(Set Int)*, but this typing would allow the protocol to, e.g., do arithmetic on node ids, which may not be desirable.) We discuss in Section III-B the use of opaque sets.

B. Protocol Semantics

A *state* of a protocol instance is an assignment of values to the variables in VARS. 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 instance containing that state. A run of a protocol instance is a possibly infinite sequence of states $s_0, s_1, s_2 \ldots$ 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. At a high-level, some transitions are called *fair* and under certain conditions, they must be taken. In this way, certain sequences of states are excluded from the set of runs of the protocol. In particular, if a sequence of states that would otherwise be a run ends in a certain cycle, that sequence may be excluded from the set of runs of a protocol due to fairness constraints.

C. Properties and Verification

We support standard temporal safety and liveness *properties* for specifying protocol correctness. Safety is often specified

```
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).

using a state *invariant*: a predicate mapping a state to true or false. A protocol instance satisfies a state invariant if all reachable states satisfy the invariant. A protocol instance satisfies a temporal property if all runs (or fair runs, if fairness is assumed) satisfy the property. A protocol satisfies a property if all its protocol instances satisfy the property.

D. 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 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*′.

In general, post-clauses may be arbitrary predicates involving the primed variable (e.g. $v' \in e$). We assume that all synthesized post-clauses are of the form $v' = e$ where e does not contain any primed variables, but make no assumptions on the post-clauses that we do not synthesize. In synthesis, non-determinism is used extensively, e.g., for modeling the environment. We note that the $v' = e$ assumption does not limit us to deterministic protocols, since multiple actions may be enabled at the same state.

Some actions are *parameterized*. For instance on line 7 of Figure 1, the action *VoteYes* is parameterized by a symbol n. From line 17, we can infer that n denotes an element of the set *Node*. The *arguments* of an action are those symbols like n. The *domain* of an argument to an action is the set quantified over for that argument in NEXT. For example, the domain of argument n of action *VoteYes* is the set *Node*. We require that the domain of an argument be a symbol from PARAMS of type *Domain*. An action may have multiple arguments; the domain of an action is the Cartesian product of the domains of its arguments. A parameterized action denotes a family of actions, one for each element in its domain.

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. We write $s \xrightarrow{A(\vec{v})} t$ to explicitly denote the arguments to A; $\vec{v}$ is empty in the case of non-parameterized actions. 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 (c.f. Section IV-C). Fairness constraints are often specified using actions: we may say A *is (strongly) fair* to mean that action A must be taken if it is enabled infinitely often.

III. SYNTHESIS OF DISTRIBUTED PROTOCOLS

A. Protocol Sketches

A tuple $\langle \text{PARAMS}, \text{VARS}, \text{HOLES}, \text{INIT}, \text{NEXT}_0 \rangle$ is a *protocol sketch*, where PARAMS, VARS, and INIT are as in a TLA+ protocol and NEXT_0 is a transition relation predicate containing the hole names found in HOLES. HOLES is a (possibly empty) set of tuples, each containing a hole name h, a list of argument symbols $\vec{v}_h$, an output type t_h, and a grammar G_h. A hole represents an uninterpreted function of type t_h 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.

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, the return type is *(Set (OfDomain Node))*, and the action of the hole is *VoteYes*(n). One grammar for this hole might be (in Backus Normal Form):

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

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 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 protocol parameters in PARAMS, the state variables in VARS, and the arguments of A_h if the action is parameterized. If h is a pre-hole, then its return type is 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*.

B. Problem Statements

A *completion* of a sketch is a protocol derived from the sketch by replacing each hole with an expression from its grammar. Informally, the synthesis task is to find a completion of the protocol that satisfies a given property. The distinction between a protocol and an instance of a protocol is important here; it may be easier to find a completion of a protocol such that a specific (e.g., finite) instance satisfies a property than to find a completion such that all instances satisfy the property. Therefore, we define two versions of the synthesis problem:

Problem 1. *Let* $\langle \text{PARAMS}, \text{VARS}, \text{HOLES}, \text{INIT}, \text{NEXT}_0 \rangle$ *be a sketch and* Φ *a property. Let* CONST *be an assignment to* PARAMS. *Find a completion,* $\langle \text{PARAMS}, \text{VARS}, \text{INIT}, \text{NEXT} \rangle$*, of the sketch such that the instance* $\langle \text{CONST}, \text{VARS}, \text{INIT}, \text{NEXT} \rangle$ *satisfies* Φ.

Problem 2. *Let* $\langle \text{PARAMS}, \text{VARS}, \text{HOLES}, \text{INIT}, \text{NEXT}_0 \rangle$ *be a sketch and* Φ *a property. Find a completion,* $\langle \text{PARAMS}, \text{VARS}, \text{INIT}, \text{NEXT} \rangle$*, of the sketch such that every instance of the completion satisfies* Φ.

In this paper we focus on solving Problem 1. It is a more tractable problem and we are able to use a model checker as a subroutine in cases where the instance has finitely many states. It turns out that in many cases, a solution to Problem 1 generalizes and is also a solution to Problem 2. This generalizability comes from the fact that the symbols in PARAMS are opaque; e.g., we may refer to the set of nodes *Node*, but we cannot refer to any particular element of *Node* without quantification. Indeed, as we show in Section V, our tool is able to synthesize protocols that generalize, i.e., they are also solutions to Problem 2.

IV. Our Approach

As mentioned in the introduction, we follow the CEGIS paradigm which includes two main components: a learner and a verifier. In our case, the learner and verifier interact in a loop with the following steps: (1) the learner generates a candidate completion X, if one exists, that satisfies a (possibly empty) set of *pruning constraints* (i.e., X is pruned if it *violates* the constraints), (2) the verifier checks X against the supplied property Φ, (3) if X satisfies Φ, a solution is found and the algorithm terminates, (4) if X does not satisfy Φ, the verifier produces a counterexample run r, (5) the learner uses r to add new pruning constraints, and we repeat until a solution is found or the search space is exhausted.

Our learner component has three subcomponents: the *expression generator* (EG), the *pruning constraint checker* (PCC), and the *counterexample generalizer* (CXG). EG generates expressions from grammars, as detailed in Section IV-A. PCC checks each generated expression against the current set of pruning constraints, as explained in Section IV-B. CXG is invoked in Step (5) to update the pruning constraints by *generalizing* the information contained in the counterexamples, as detailed in Section IV-C.

Pruning constraints eliminate candidate completions that are guaranteed to exhibit previously encountered counterexamples, without having these candidates checked by the verifier, which is often an expensive subroutine. A naive way to do that would be to keep a list L of counterexamples seen so far, and then check whether a candidate exhibits any of the runs in L. Instead, we use more sophisticated pruning constraints that encode counterexamples as logical constraints on uninterpreted functions, c.f. Sections IV-B and IV-C.

As our verifier in Step (2), we use an off-the-shelf TLA$^+$ model checker, specifically TLC [51]. We will not discuss TLA$^+$ model checking further as it is standard.

A. Expression Generation

Recall that candidate protocols are completions of some sketch, which are in turn charaterized as members of some grammar. In the case of multi-hole sketches, completions are characterized as members of the cross-product of the grammars. Therefore, generating candidate protocols reduces to *enumerating* expressions from grammars. Note that, although grammars have a finite representation, the *language* (i.e., set of expressions) of a grammar may be infinite and the expressions therein may be arbitrarily large.

We experimented with three grammar enumeration techniques: (1) a naive breadth-first algorithm, (2) a cache-based algorithm, and (3) an extension of the cache-based algorithm that exploits semantic equivalence of expressions.

1) Naive Breadth-First Algorithm: A naive breadth-first search algorithm is to keep a priority queue (sorted by size) of *partial expressions*, i.e. expressions containing both terminals and non-terminals. A partial expression is discharged from the queue by considering all possible ways to replace the non-terminals using the grammar rules and substituting those into the partial expression. For example, if the partial expression is $d := E \cup (x \cup E)$ and the grammar has a production $G := E ::= x \mid y \mid E \cup E$, we would consider nine different partial expressions, one for each pair of productions of the E rule, since E appears twice in the expression d. After substituting, we can immediately return those expressions which do not contain non-terminals and add to the queue those that do. Our experience was that this algorithm was too slow in practice, since it iterates over and performs substitutions on larger and larger partial expressions.

2) Cache-Based Algorithm: In the cache-based algorithm, our learner generates all candidates of size n before it generates any candidates of size $n + 1$. Expressions are essentially trees and our notion of *size* is the number of nodes in the tree, e.g., the size of $(a + b) + c$ is 5. We keep a cache mapping each integer n to the set of all non-partial expressions of that size, for each non-terminal. We then use this cache to build larger non-partial expressions, substituting only into productions (partial expressions) that appear in the grammar. There are often many expressions of size n. We use generators

to yield a stream of expressions, which avoids generating all expressions of a given size at once.

As an example of the cache-based algorithm, suppose we want to generate the expressions of size 5 for the non-terminal E in the grammar G above. Assume we already have a cache containing all expressions of size 1,2,3, and 4 for E. Then we can generate all expressions of size 5 by substituting pairs of expressions into the rule $E ::= E \cup E$ such that the sum of the sizes of the two expressions is 5.

3) Equivalence Reduction: Because the cache-based algorithm reuses all expressions of a given size many times over, it is important to keep the cache as small as possible. In particular, if two expressions are *semantically equivalent*, only one should appear in the cache. To illustrate, consider that there are only 16 boolean expressions over two variables, modulo equivalence. The grammar $B ::= x \mid y \mid \neg B \mid B \wedge B$ can express all 16 of these expressions, but it generates infinitely many expressions. The number of expressions of size n is $O(2^n)$.

When we generate a new expression, we compute a normal form for that expression. We then check if we have already generated an expression with that normal form. If we have, we do not return the new expression and we do not add it to the cache. We implement normal forms for (1) set expressions containing the operations $\cup$, $\cap$, and $\setminus$, (2) boolean expressions containing the operations $\vee$, $\wedge$, and $\neg$, (3) equality expressions, and (4) inequality expressions. We use DNF as the normal form for boolean expressions. Our normal form for set expressions exploits the correspondence between set expressions and boolean functions and then uses DNF. Equality between sets A and B is equivalent to $\emptyset = (A \setminus B) \cup (B \setminus A)$; we exploit this fact to obtain a normal form for equality between two sets.

In addition to equivalence reduction by normal forms, we also exploit the semantics of expressions to *short-circuit* the generation of expressions. Short-circuiting is a technique that allows us to avoid iterating over large parts of the search space. For instance, if we are generating expressions of size 5 for the rule $E \cup E$, we can consider pairs of expressions of sizes (1,4) and (2,3), but we can exploit the commutativity of union by ignoring sizes (4,1) and (3,2). Without this technique, we would have to iterate over twice as many pairs of expressions, compute their normal forms, and check if these normal forms are in the cache. In general, for commutative operation $\odot$ if we are generating expression $e_1 \odot e_2$, we first pick e_1 and only iterate over choices for e_2 that are at least as large as e_1.

B. Counterexamples and Pruning Constraints

1) Counterexamples: A counterexample is a run of the protocol annotated with actions: $s_0 \xrightarrow{A_1(\vec{v_1})} s_1 \xrightarrow{A_2(\vec{v_2})} \ldots \xrightarrow{A_k(\vec{v_k})} s_k$. The run is reported as a safety, deadlock, or liveness violation. If the run is a safety or deadlock violation, it is interpreted as a path. If it is a liveness violation, it is interpreted as a path leading to a cycle, called a *lasso*. In the case of liveness, $s_k = s_i$ for some $i < k$ and s_i is the state that first injects into the cycle.

2) Pruning Constraints: Our pruning constraints are logical constraints over propositional logic with equality and uninterpreted functions. For example, suppose we have the holes $h_1(a, b)$ and $h_2(b, c)$. Then, an example of a pruning constraint is the formula $\pi := (h_1(0, 1) \neq \textit{True}) \vee (h_2(1, 2) \neq 1)$. π constrains the candidate expressions for the holes h_1 and h_2. For example, replacing h_1 and h_2 with the expressions $a < b$ and $c - b$, respectively, violates π, because $0 < 1 = \textit{True}$ and $2 - 1 = 1$. Replacing h_1 and h_2 with $a < b$ and b, respectively, also violates π. Hence, π prunes at least two completions.

Formally, a pruning constraint is a disjunction of *terms*, where each term is a triple containing (1) a hole h, (2) a mapping $s^\star$ from the arguments of h to values, and (3) a literal value of the output type of h. For instance, in π above, the first term has $h = h_1(a, b)$, $s^\star = [a \mapsto 0, b \mapsto 1]$, and $y = \textit{True}$. The second term has $h = h_2(b, c)$, $s^\star = [b \mapsto 1, c \mapsto 2]$, and $y = 1$. Let $\tau := (h, s^\star, y)$ be a term and let $\widehat{h}$ be an interpretation (in our case, an expression) for the uninterpreted function h. Then $\widehat{h}$ satisfies τ if $\widehat{h}(s^\star) \neq y$. If $h_1, h_2, ..., h_m$ are the uninterpreted functions in a pruning constraint π and $X := \widehat{h}_1, \widehat{h}_2, ..., \widehat{h}_m$ are interpretations for the h_i (i.e. a completion), then X satisfies π if the disjunction of the τ terms in π is satisfied.

In each run, our algorithm maintains a *set* of pruning constraints, interpreted as a *conjunction* (of disjunctions of terms). A completion satisfies a set of pruning constraints if it satisfies all constraints in the set. Because we want to avoid seeing any counterexample more than once, the learner will pass a completion X to the verifier only if X satisfies every pruning constraint. I.e., a pruning constraint π prunes completions that do *not* satisfy π. PCC checks against the pruning constraints by substituting the expressions of the holes into the constraints and performing evaluation. Each type of counterexample (safety, deadlock, liveness) requires a slightly different encoding as a pruning constraint, as explained next.

C. Counterexample Generalization

A pruning constraint π is *under-pruning* w.r.t. run r and sketch S if there exists a completion X of S such that X satisfies π and r is a run of X. π is *over-pruning* w.r.t. run r and sketch S if there exists a completion X of S such that X does not satisfy π and r is not a run of X. π is *optimal* if it is neither under- nor over-pruning. π is *sub-optimal* if it is under-pruning, but not over-pruning. Our primary goal is to avoid over-pruning constraints, since over-pruning results in an incomplete algorithm, i.e., an algorithm that might miss valid completions.

In what follows we present three techniques to encode into pruning constraints, safety, deadlock, and liveness counterexamples, respectively. Our safety pruning constraints are optimal (Theorem 2), but our deadlock and liveness pruning constraints are sub-optimal (Theorems 3 and 4). In practice, these sub-optimal constraints are sufficient to avoid many completions that exhibit the corresponding violations; the bottleneck in our experiments is not the number of model checker calls.

1) Encoding Safety Counterexamples: Intuitively, a safety violation can be fixed by "cutting" at least one transition in the counterexample run, either by violating its guard or by modifying its state update. Let $r = s_0 \xrightarrow{A_1(\vec{v_1})} s_1 \xrightarrow{A_2(\vec{v_2})} \ldots \xrightarrow{A_k(\vec{v_k})} s_k$ be a safety violation and suppose that the completion is characterized by the interpretations $\widehat{h}_1, \widehat{h}_2, ...\widehat{h}_m$. We denote the pruning constraint for r as $\pi_{\textit{safe}}(r)$ and construct it as follows. $\pi_{\textit{safe}}(r)$ is a disjunction of τ*-terms*. For each $s \xrightarrow{A(\vec{v})} t$ in the counterexample, we construct a set of τ-terms. In particular, for each hole h_i in the action A, we construct the term $\tau_{A(\vec{v}),i} := (h_i, s^\star, y)$, where $y := \widehat{h}_i(s^*)$ and where $s^\star$ is the predecessor state s, restricted to the arguments of h_i, including the arguments to the action A. The pruning constraint is then the disjunction containing all $\tau_{A(\vec{v}),i}$.

For instance, suppose the safety violation is $[a, b, c \mapsto 0, 1, 2] \xrightarrow{A} [a, b, c \mapsto 1, 1, 2]$. Suppose additionally that $h_1(a, b)$ is a pre-hole in A and $a' = h_2(b, c)$ is a post-hole in A. Suppose that the completion that resulted in the safety violation had $\widehat{h}_1(0, 1) = \textit{True}$ and $\widehat{h}_2(1, 2) = 1$. Then $\tau_{A,1} = (h_1, [a \mapsto 0, b \mapsto 1], \textit{True})$ and $\tau_{A,2} = (h_2, [b \mapsto 1, c \mapsto 2], 1)$. The pruning constraint would be $\tau_{A,1} \vee \tau_{A,2}$, which corresponds to π from before. This constraint ensures that the pre-condition of A is not satisfied in the state $[a, b, c \mapsto 0, 1, 2]$ or that $a \neq 1$ after taking action A in that state.

2) Encoding Deadlock Counterexamples: Informally, a pruning constraint of a deadlock violation is similar to that of a safety violation because a deadlock violation can be fixed by making the deadlocked state s_k unreachable. But another way to fix a deadlock violation is to make s_k *undeadlocked*, which may be done by weakening the pre-condition of some action that is not enabled in s_k.

Formally, the deadlock pruning constraint for run r is defined to be $\pi_{\textit{dead}}(r) := \pi_{\textit{safe}}(r) \vee \pi_\rho(r)$, where $\pi_\rho(r)$ is a disjunction of ρ*-terms*, each of the form $\rho_{A(\vec{v}),i,k} := (h_i, s_k^\star, y)$, where $s_k^\star$ is s_k restricted to the arguments of h_i and where $y := \widehat{h}_i(s_k^\star)$. We construct a ρ-term for every action A and every pre-hole h_i in A such that $\widehat{h}_i(s_k) = \textit{False}$. Then $\pi_\rho(r)$ is the disjunction of all all ρ-terms.

3) Encoding Liveness Counterexamples: The constraint for a liveness violation can be thought of as a generalization of the constraint for a deadlock violation. It is sufficient to do one of (1) break the path to the cycle using τ-terms, (2) break the cycle using τ-terms, or (3) weaken the pre-condition of some fair action that is not enabled in some state of the cycle using ρ-terms, making the cycle *unfair*. Formally, we denote our liveness pruning constraint as $\pi_{\textit{live}}(r)$. We construct it as $\pi_{\textit{live}}(r) := \pi_{\textit{safe}}(r) \vee \pi'_\rho$, where π'_ρ is the disjunction of the following ρ-terms: For each fair action A, for every $\vec{v}$ in the domain of A, for every j such that s_j is in the cycle, and for every *pre-hole* h_i in A such that $\widehat{h}_i(s_j) = \textit{False}$, we construct the term $\rho_{A(\vec{v}),i,j}$.

4) Fairness and Stuttering: Although we are able to handle both weakly and strongly fair actions, we did not treat them differently above in $\pi_{\textit{live}}$. That construction may be under-pruning in the presence of weakly fair actions, but it will never over-prune and therefore our algorithm is complete. None of our benchmarks required weak fairness when modeling the synthesized protocols.

Stuttering (a special liveness violation) occurs when there are no fair, enabled, non-self-looping actions in the final state of the violation. In constrast, deadlock violations occur when there is no enabled action at all. We denote the pruning constraint for a stuttering violation as $\pi_{\textit{stut}}(r) := \pi_{\textit{safe}}(r) \vee \pi_\tau \vee \pi'_\rho$. In addition to the τ-terms from $\pi_{\textit{safe}}(r)$, we add π_τ, which is the disjunction of $\tau_{A(\vec{v}),i}$ for every post-hole h_i in every fair action A. We add π'_ρ as we did for a typical liveness violation, except the only s_j in the cycle is the last state of r, s_k.

Theorem 1. *Let r be a counterexample of a completion of the sketch S. If r is a safety violation then $\pi_{\textit{safe}}(r)$ is optimal w.r.t. r and S. If r is a deadlock, liveness, or stuttering violation then $\pi_{\textit{dead}}(r)$, $\pi_{\textit{live}}(r)$, and $\pi_{\textit{stut}}(r)$, respectively, are sub-optimal w.r.t. r and S.* — The proof can be found in Appendix A.

V. Implementation and Evaluation

Implementation and Experimental Setup: We implemented our method (Section IV) in a tool, Scythe, which supports many features of the TLA$^+$ language and utilizes the TLC model checker [51] as verifier. Scythe is written in Python and takes as input (1) a TLA+ file defining the protocol and its sketch and (2) a configuration file defining the grammars and types along with protocol parameters. Our grammars are typed regular tree grammars [15] and our implementation essentially uses the standard SYNTH-LIB input format for SyGuS [1]. We ran each experiment on a dedicated 2.40 GHz CPU.

Benchmarks: Our benchmark suite contains seven distinct protocols: (1) decentralized lock service (decentr. lock), (2) server-client lock service (lock_serv), (3) Peterson's algorithm for mutual exclusion, (4) two phase commit (2PC), (5) consensus, (6) sharded key-value store (sharded_kv), (7) raft-reconfig, and (8) raft-reconfig-big. (7) and (8) are non-trivial, reconfigurable variants of the Raft protocol [33], [40], [41]. Our benchmarks are adapted from safety verification benchmarks that have been used in recent years [41], [17]. These existing benchmarks contain a suite of correct, manually crafted protocols and we refer to each manually crafted solution as the *ground truth*. We report statistics about the ground truth for reference, but we do not use this information during synthesis. For instance, we do not assume knowledge of which variables a missing expression depends on.

Adapting verification benchmarks for synthesis by sketching requires a number of steps, some of which are non-trivial. We discuss the most salient points of these steps next.

Holes: For each protocol we performed many synthesis experiments by varying the number of holes in the protocol sketch. All our experiments, as well as instructions for reproducing them, can be found on GitHub [13]. Representative experiments are summarized in Table I, explained below.

Grammars: Each hole requires a grammar. Scythe is flexible; the user can provide a different grammar for each hole, or reuse grammars across holes. Scythe grammars are

modular in the sense that they contain a *general-purpose* part (e.g., the grammar of boolean or arithmetic or set expressions) plus a *hole-specific* part (e.g., the terminals which are the hole's arguments). We implemented a library that allows to build grammars by (1) automatically constructing non-terminals based on the types of the hole's arguments and (2) exposing to the user common sub-grammars that can be deployed across protocols.

Liveness and Fairness: Our benchmarks come from existing suites focusing on *safety* verification [41], [17]. Performing synthesis against only safety properties often results in *vacuous* solutions that satisfy safety in trivial ways (e.g. by filling a pre-hole with the expression *False*). Therefore, we augment each benchmark with additional liveness properties and any necessary fairness constraints.

Implementability Constraints: In addition to excluding vacuous solutions by adding extra properties, we sometimes need to exclude *unimplementable* solutions, for instance, solutions violating implicit communication/observability constraints between the protocol processes. For example, replacing the post-condition in line 8 of Fig. 1 with $vote\_yes' = \emptyset$ results in an unimplementable protocol because a node cannot directly change the vote state of another node. To avoid such solutions, we used arrays instead of sets (e.g., *vote_yes* is an array mapping process ids to booleans). Then, we restricted the grammar to only contain array access expressions with appropriate indices.

Explicitly Modeling the Environment: We had to modify several of the verification benchmarks of [41] in order to explicitly separate the (controllable) protocol from its (uncontrollable) environment, so as to prevent synthesis of parts belonging to the environment.

Results: TLA$^+$ LOC is the number of lines of code of the ground truth TLA$^+$ protocol specification, which is the same as the lines of code in the sketch and the synthesized protocols, since all synthesized expressions are printed to one line, regardless of size. ID refers to the number used to identify the experiment in the full results table [13]. "#pre/post holes" is the number of pre- and post-holes in the sketch, and k refers to $k = k_1 + k_2 + ... + k_n$, where each k_i is the *size* of the expression (c.f. Section IV-A) used in the ground truth protocol for the ith hole. "gram. LOC" is the number of lines (non-boilerplate) code in the python script used to generate the grammar. Every protocol uses the same grammar generation script, regardless of which or how many holes are poked.

We report Execution Statistics for the tool with and without equivalence reduction. The column "generated / model checked" reports the number of completions generated by the tool vs those model checked (the rest were pruned). The column k' is either the size of the expression found by the tool, or the size of the largest expression the tool considered before it timed out (marked with a $\geq$ symbol). If there are multiple holes then k' is the sum of all expression sizes. Column "total / model checking time" reports the total execution time vs the time devoted to model checking (both in seconds). TO indicates that the tool timed out after 1 hour; TO** is explained below. Note that TLC is called without a timeout; hence, it performs exhaustive model checking on the finite protocol instances specified by the user configuration.

As Table I shows, our efficient expression generation technique with equivalence reduction achieves impressive results, sometimes reducing the number of generated expressions by more than three orders of magnitude (c.f. raft-reconfig ID 121 where only 271 expressions are generated with reduction, vs $>$ 690,000 without reduction at the TO point). In all cases, the number of completions model checked is much smaller than those generated, which shows how critical pruning constraints are to scalability. With equivalence reduction, execution time is typically dominated by model checking, although there are exceptions (e.g. 2pc). Without equivalence reduction, the time is typically dominated by expression generation, which demonstrates the importance of the equivalence reduction.

Qualitatively, SCYTHE often synthesizes large, non-trivial expressions, e.g., single expressions of size 14 in the cases of raft-reconfig ID 121 and raft-reconfig-big ID 714, and multiple expressions of combined size up to 18 in other cases. Expressions (1) and (2) shown in Section I are two concrete examples of synthesized expressions.

Novel Solutions: The protocols synthesized by SCYTHE were often identical (or almost identical, up to commutativity of an operator such as $\wedge$, etc.) to the ground truth. In other cases, however, SCYTHE found novel, non-vacuous solutions. SCYTHE often found solutions with shorter expressions. One notable example comes from the experiment 2pc ID 303, where instead of the ground-truth expression $e_1 := \emptyset \neq (P \setminus A) \cup (P \cap N)$, SCYTHE found the expression $e_2 := P \neq (A \setminus N)$, where P is the set of all nodes, A is the set of alive nodes, and N is the set of nodes that voted no. So e_1 says "There is a node that is dead or there is at least one node that voted no." In the context of the protocol, e_1 and e_2 are equivalent, but a proof requires the subtle reasoning that both A and N are subsets of P, since P is the set of all nodes.

Correctness of Infinite Instances: A protocol synthesized by SCYTHE is a solution to Problem 1, i.e., is correct for the finite instance specified by the user. This correctness follows from the fact that during the synthesis loop the verifier (TLC) exhausts the state space of the specified finite instance. As it turns out, the protocols of Table I produced by SCYTHE are also solutions to Problem 2. Specifically, for each protocol of Table I except peterson, we used the TLA$^+$ Proof System (TLAPS) [11] to prove that all instances of that protocol satisfy the key safety property (our TLAPS proofs did not consider liveness; the four peterson variants involve only two processes and need no extra verification). For our TLAPS proofs we used techniques similar to those reported in [40].

In all but two cases, the initial solutions produced by SCYTHE proved to be correct. For raft-reconfig ID 343 and raft-reconfig-big ID 709, SCYTHE initially produced a solution which is correct for up to 3 nodes, but which we were surprised to find is incorrect for 4 or more nodes. To address this scenario, we added to the tool an *extra-check* option to perform an additional model checking step with larger parameter values

		Sketch Parameters				Execution Stats					
						w/o eq. reduction			w/ eq. reduction		
Protocol	TLA$^+$ LOC	ID	#pre/post holes	k	gram. LOC	**generated/** model checked	k'	**total** / model checking time	**generated/** model checked	k'	**total** / model checking time
decentr. lock	48	486	1 / 3	21	28	**93403** / 136	16	**674** / 193	**3020** / 117	*16*	**190** / 175
lock_serv	83	599	2 / 6	16	22	**384569** / 85	16	**2116** / 134	**7483** / 80	*16*	**159** / 120
lock_serv	83	611	2 / 6	16	22	**665463** / 104	$\geq$16	**TO** / 1094	**4064** / 84	*16*	**145** / 124
peterson	105	475	3 / 1	19	53	**442553** / 324	12	**2731** / 453	**485** / 243	*12*	**348** / 346
peterson	105	375	2 / 2	19	53	**583201** / 264	13	**3355** / 356	**1369** / 267	*13*	**364** / 357
peterson	105	413	3 / 1	22	53	**582616** / 353	$\geq$12	**TO** / 602	**7073** / 1259	*15*	**1809** / 1753
peterson	105	547	2 / 6	22	53	**643529** / 167	$\geq$16	**TO** / 483	**5569** / 222	*17*	**329** / 301
2pc	134	303	2 / 0	15	46	**690411** / 24	$\geq$8	**TO** / 1494	**65994** / 23	*9*	**388** / 43
2pc	134	558	3 / 5	18	46	**410675** / 87	14	**2301** / 179	**89027** / 88	*14*	**629** / 171
2pc	134	485	2 / 2	17	46	**681190** / 44	$\geq$10	**TO** / 492	**493492** / 55	*11*	**2654** / 110
2pc	134	513	2 / 6	18	46	**642178** / 162	$\geq$18	**TO** / 1641	**98009** / 211	*18*	**886** / 382
consensus	127	624	2 / 6	17	56	**550501** / 97	17	**3442** / 606	**9988** / 63	*17*	**427** / 375
consensus	127	550	2 / 6	22	56	**483126** / 162	$\geq$17	**TO** / 1116	**53994** / 286	*18*	**2291** / 2011
sharded_kv	112	302	1 / 1	13	44	**248298** / 13	13	**1325** / 49	**469** / 14	*13*	**59** / 57
sharded_kv	112	365	1 / 3	22	44	**611512** / 129	$\geq$16	**TO** / 941	**3149** / 149	*17*	**472** / 455
raft-reconfig	174	463	1 / 3	21	82	**64832** / 64	14	**462** / 128	**1958** / 65	*14*	**139** / 129
raft-reconfig	174	343	2 / 0	21	82	**608586** / 215	$\geq$11	**TO** / 484	**41411** / 251	*17*	**750** / 530
raft-reconfig	174	121	1 / 0	18	82	**694552** / 12	$\geq$12	**TO** / 3589	**271** / 13	*14*	**31** / 27
raft-reconfig-big	304	708	1 / 3	21	85	**67237** / 78	14	**1815** / 1465	**3155** / 84	*14*	**1668** / 1651
raft-reconfig-big	304	709	2 / 0	21	85	**2231397** / 220	$\geq$12	**TO**** / 3950	**282106** / 252	*17*	**TO**** / 12943
raft-reconfig-big	304	710	1 / 0	18	85	**658492** / 12	$\geq$12	**TO** / 3588	**1369** / 13	*14*	**368** / 359
raft-reconfig-big	304	714	1 / 7	25	85	**221648** / 101	18	**2662** / 1519	**6500** / 102	*18*	**1802** / 1768

TABLE I

than those used in the synthesis loop, right before outputting the final solution (if the extra-check fails, the tool continues to search for a solution). SCYTHE with extra-check found a correct (for all instances) solution for raft-reconfig ID 343 in 750 secs (this includes the time spent for extra-checks). SCYTHE with equivalence redution also found a correct (for all instances) solution for raft-reconfig-big ID 709, although it timed out after a total of four hours (TO**) while performing the final extra-check for 4 nodes—that single final extra-check took about 2 hours. For ID 709, SCYTHE without equivalence reduction failed to find a solution as it spent 4 hours generating expressions that were much smaller ($\leq$ size 12) than the solution found with equivalence reduction (size 17).

VI. Related Work

Past works synthesize explicit-state, finite-state machines [2], [5], [14], [16]. In contrast, we synthesize symbolic and parameterized infinite-state machines. TRANSIT [47] cannot process counterexamples automatically and requires a human in the synthesis loop. [30] and [6] use *cut-off* techniques which only apply to a special class of self-stabilizing protocols in symmetric networks, and [24] study a special class of distributed agreement-based systems. [29] consider only threshold-guarded distributed protocols. In contrast, our work applies to general distributed protocols.

As discussed in Section I, [4] synthesize interpretations of uninterpreted functions represented as finite lookup tables, whereas we synthesize symbolic expressions directly. We use TLA$^+$ models with parameterized actions. In contrast, [4] use extended finite state machines (EFSMs) which do not have parameterized actions. It is unclear whether expressions such as (1) and (2) on page 282 could be synthesized by [4].

Like [4], we use CEGIS and our counterexample encodings are similar. Unlike [4], we rely neither on an external SyGuS solver nor on an SMT solver. [4] encodes the search space of candidate interpretations as SMT formulas and calls an SMT solver to generate the next candidate. SMT queries are both expensive and numerous in the context of CEGIS. In contrast, we use efficient grammar enumeration techniques and we bypass SMT solvers by checking candidate expressions directly against the pruning constraints (Section IV).

Like [4], our tool synthesizes solutions that are guaranteed correct only up to the finite instances model checked in the CEGIS loop. Unlike [4], we went one step further and proved with TLAPS that the solutions produced by our tool are actually correct for all instances. As discussed in Section V, this step is not redundant: there were surprising cases of solutions which are correct for 3 nodes but not for 4 or more nodes. It is unclear whether the protocols synthesized in [4] are correct beyond the finite model checked instances.

[26] use genetic programming and [23] use machine learning for synthesis. Generally, these approaches are not guaranteed to find a solution even if one exists, i.e. they are incomplete. In contrast, our approach is complete.

None of the works cited above use syntax to guide the search, none use equivalence of expressions with short-circuiting to reduce the search space, and none handle state variables with infinite domains. To our knowledge, ours is the only truly syntax-guided synthesis method for symbolic, parameterized distributed protocols.

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 [38] 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. The "fast term enumeration strategy" of cvc4sy is similar to our cache-based approach and also uses equivalence reduction techniques. To our knowledge, cvc4sy does not use short-circuiting.

In our work, we assume that the user has a means of constructing the appropriate sketch; we do not address the problem of "sketch inference." Work on scenarios [2] and flows [44] is directly applicable to this problem of coming up with a sketch. The sketch may also arise from a manually constructed, incorrect protocol that the user wishes to *repair* [7], along with knowledge of where a bug exists. Likewise, we assume that the user provided a sketch-property pair that is *realizable*—i.e., there exists a completion of the sketch that satisfies the property. Recent work on *unrealizability logic* [22], [27], [31] provides insight on how to identify unrealizable synthesis instances and communicate appropriate information to the user to help facilitate sketch debugging. A synthesis pipeline that integrates our work with that above is a promising direction for future work.

VII. Conclusion

We present the only, to our knowledge, truly syntax-guided synthesis method for symbolic, parameterized, infinite-state distributed protocols. We show experimentally that our method and tool are able to synthesize non-trivial completions across a broad set of non-trivial protocols written in TLA$^+$, and prove that these completions generalize correctly (i.e., preserve safety) in all possible instances.

Our sketch-based approach to distributed protocol synthesis is motivated by several factors. First, a common pattern in the design of distributed protocols is to extend an existing protocol (e.g. a non-reconfigurable protocol) with a new feature (e.g. reconfiguration). Indeed, our benchmarks include variants of the Raft dynamic reconfiguration protocol [33], [40], [41], and we focus on synthesizing the "Reconfig" action of those protocols. Sketching naturally fits this design pattern. Second, if a bug has been localized to a specific part of a protocol, sketching can be used to repair the protocol [7]. Finally, synthesis "from scratch" is a special case of synthesis by sketching where the sketch admits all protocols as completions. Therefore, no generality is lost when studying a sketch-based approach to synthesis and tractability is gained.

Future work includes: (1) further ways to reduce the search space and short-circuit parts of the search; (2) optimization of the Scythe-TLC interface to avoid running a new instance of (and repeatedly initializing) TLC each time Scythe needs to check a candidate protocol; (3) addressing the problems of sketch inference and unrealizability handling for the synthesis of distributed protocols; and (4) automating the final, all-instances verification step (generally an undecidable problem), by potentially combining TLAPS with state of the art inductive invariant inference techniques [39].

References

[1] Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, pages 1–8. IEEE, 2013.

[2] Rajeev Alur, Milo Martin, Mukund Raghothaman, Christos Stergiou, Stavros Tripakis, and Abhishek Udupa. Synthesizing Finite-state Protocols from Scenarios and Requirements. In *Haifa Verification Conference*, volume 8855 of *LNCS*. Springer, 2014.

[3] Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. Scaling enumerative program synthesis via divide and conquer. In *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017*, volume 10205 of *Lecture Notes in Computer Science*, pages 319–336, 2017.

[4] Rajeev Alur, Mukund Raghothaman, Christos Stergiou, Stavros Tripakis, and Abhishek Udupa. Automatic completion of distributed protocols with symmetry. In Daniel Kroening and Corina S. Pasareanu, editors, *Computer Aided Verification - 27th International Conference, CAV*, volume 9207 of *Lecture Notes in Computer Science*, pages 395–412. Springer, 2015.

[5] Rajeev Alur and Stavros Tripakis. Automatic synthesis of distributed protocols. *SIGACT News*, 48(1):55–90, 2017.

[6] Roderick Bloem, Nicolas Braud-Santoni, and Swen Jacobs. Synthesis of self-stabilising and byzantine-resilient distributed systems. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification - 28th International Conference, CAV*, volume 9779 of *Lecture Notes in Computer Science*, pages 157–176. Springer, 2016.

[7] Borzoo Bonakdarpour and Sandeep S. Kulkarni. Automated model repair for distributed programs. *SIGACT News*, 43(2):85–107, 2012.

[8] Ethan Buchman. Tendermint: Byzantine fault tolerance in the age of blockchains. 2016.

[9] Vitalik Buterin. Ethereum white paper: A next generation smart contract & decentralized application platform. 2013.

[10] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):1–22, 2013.

[11] Denis Cousineau, Damien Doligez, Leslie Lamport, Stephan Merz, Daniel Ricketts, and Hernan Vanzetto. TLA+ Proofs. *18th International Symposium on Formal Methods (FM 2012)*, 7436:147–154, January 2012.

[12] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. *ACM SIGOPS operating systems review*, 41(6):205–220, 2007.

[13] Derek Egolf. scythe-fmcad2024. https://github.com/egolf-cs/scythe-fmcad2024.

[14] Derek Egolf and Stavros Tripakis. Synthesis of distributed protocols by enumeration modulo isomorphisms. In *ATVA 2023 - Part I*, Lecture Notes in Computer Science, pages 270–291. Springer, 2023.

[15] Joost Engelfriet. Tree automata and tree grammars. *CoRR*, abs/1510.02036, 2015.

[16] Bernd Finkbeiner and Sven Schewe. Bounded synthesis. *Int. J. Softw. Tools Technol. Transf.*, 15(5-6):519–539, 2013.

[17] Aman Goel. IvyBench. https://github.com/aman-goel/ivybench, Accessed: 2024-04-22.

[18] Aman Goel and Karem Sakallah. On Symmetry and Quantification: A New Approach to Verify Distributed Protocols. In *NASA Formal Methods: 13th International Symposium, NFM 2021*, page 131–150, 2021.

[19] Aman Goel and Karem A. Sakallah. Towards an automatic proof of lamport's paxos. In *2021 Formal Methods in Computer Aided Design (FMCAD)*, pages 112–122, 2021.

[20] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. Program synthesis. *Foundations and Trends in Programming Languages*, 4(1-2):1–119, 2017.

[21] Travis Hance, Marijn Heule, Ruben Martins, and Bryan Parno. 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)*, pages 115–131. USENIX Association, April 2021.

[22] Qinheping Hu, John Cyphert, Loris D'Antoni, and Thomas W. Reps. Exact and approximate methods for proving unrealizability of syntax-guided synthesis problems. In Alastair F. Donaldson and Emina

Torlak, editors, *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, pages 1128–1142. ACM, 2020.

[23] Yujie Hui, Drew Ripberger, Xiaoyi Lu, and Yang Wang. Learning distributed protocols with zero knowledge. In *Machine Learning for Systems at NeurIPS 2023*, 2023.

[24] Nouraldin Jaber, Christopher Wagner, Swen Jacobs, Milind Kulkarni, and Roopsha Samanta. Synthesis of distributed agreement-based systems with efficiently-decidable verification. In *TACAS 2023*, volume 13994 of *Lecture Notes in Computer Science*, pages 289–308. Springer, 2023.

[25] Swen Jacobs and Roderick Bloem. Parameterized synthesis. *Log. Methods Comput. Sci.*, 10(1), 2014.

[26] Gal Katz and Doron Peled. Synthesizing solutions to the leader election problem using model checking and genetic programming. In *Haifa Verification Conference*, HVC'09, page 117–132. Springer, 2009.

[27] Jinwoo Kim, Loris D'Antoni, and Thomas W. Reps. Unrealizability logic. *Proc. ACM Program. Lang.*, 7(POPL):659–688, 2023.

[28] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, Jun 2002.

[29] Marijana Lazic, Igor Konnov, Josef Widder, and Roderick Bloem. Synthesis of distributed algorithms with parameterized threshold guards. In *21st International Conference on Principles of Distributed Systems, OPODIS*, volume 95 of *LIPIcs*, pages 32:1–32:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017.

[30] Nahal Mirzaie, Fathiyeh Faghih, Swen Jacobs, and Borzoo Bonakdarpour. Parameterized synthesis of self-stabilizing protocols in symmetric networks. *Acta Informatica*, 57(1-2):271–304, 2020.

[31] Shaan Nagy, Jinwoo Kim, Loris D'Antoni, and Thomas W. Reps. Automating unrealizability logic: Hoare-style proof synthesis for infinite sets of programs. *CoRR*, abs/2401.13244, 2024.

[32] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. How Amazon Web Services Uses Formal Methods. *Commun. ACM*, 58(4):66–73, March 2015.

[33] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319. USENIX Association, June 2014.

[34] Oded Padon, Giuliano Losa, Mooly Sagiv, and Sharon Shoham. Paxos Made EPR: Decidable Reasoning about Distributed Protocols. *Proc. ACM Program. Lang.*, 1(OOPSLA), Oct 2017.

[35] Oded Padon, Kenneth L. McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. Ivy: Safety Verification by Interactive Generalization. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 614–630. ACM, 2016.

[36] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, page 179–190, New York, NY, USA, 1989. Association for Computing Machinery.

[37] A. Pnueli and R. Rosner. Distributed reactive systems are hard to synthesize. In *Proceedings of the 31th IEEE Symposium on Foundations of Computer Science*, pages 746–757, 1990.

[38] Andrew Reynolds, Haniel Barbosa, Andres Nötzli, Cesare Tinelli, and Clark Barrett. CVC4SY: Smart and fast term enumeration for syntax-guided synthesis. In Isil Dillig and Serdar Tasiran, editors, *Proceedings of the 31st International Conference on Computer Aided Verification (CAV)*, volume 11561 of *Lecture Notes in Computer Science*, pages 74–83. Springer, July 2019.

[39] William Schultz, Edward Ashton, Heidi Howard, and Stavros Tripakis. Scalable, Interpretable Distributed Protocol Verification by Inductive Proof Slicing. arXiv eprint 2404.18048, 2024.

[40] William Schultz, Ian Dardik, and Stavros Tripakis. Formal verification of a distributed dynamic reconfiguration protocol. In *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2022, page 143–152. ACM, 2022.

[41] William Schultz, Ian Dardik, and Stavros Tripakis. Plain and Simple Inductive Invariant Inference for Distributed Protocols in TLA$^+$. In *22nd Formal Methods in Computer-Aided Design, FMCAD 2022*, pages 273–283. IEEE, 2022.

[42] Armando Solar-Lezama. The sketching approach to program synthesis. In *Proceedings of the 7th Asian Symposium on Programming Languages and Systems*, APLAS '09, pages 4–13. Springer, 2009.

[43] Armando Solar-Lezama. Program sketching. *Int. J. Softw. Tools Technol. Transf.*, 15(5-6):475–495, oct 2013.

[44] Murali Talupur, Sandip Ray, and John Erickson. Transaction flows and executable models: Formalization and analysis of message passing protocols. In Roope Kaivola and Thomas Wahl, editors, *Formal Methods in Computer-Aided Design, FMCAD 2015, Austin, Texas, USA, September 27-30, 2015*, pages 168–175. IEEE, 2015.

[45] John G. Thistle. Undecidability in decentralized supervision. *Systems & Control Letters*, 54(5):503–509, 2005.

[46] Stavros Tripakis. Undecidable Problems of Decentralized Observation and Control on Regular Languages. *Information Processing Letters*, 90(1):21–28, April 2004.

[47] Abhishek Udupa, Arun Raghavan, Jyotirmoy V. Deshmukh, Sela Mador-Haim, Milo M. K. Martin, and Rajeev Alur. TRANSIT: specifying protocols with concolic snippets. In Hans-Juergen Boehm and Cormac Flanagan, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 287–296. ACM, 2013.

[48] Jianan Yao, Runzhou Tao, Ronghui Gu, and Jason Nieh. DuoAI: Fast, Automated Inference of Inductive Invariants for Verifying Distributed Protocols. In Marcos K. Aguilera and Hakim Weatherspoon, editors, *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2022)*, pages 485–501. USENIX Association, 2022.

[49] Jianan Yao, Runzhou Tao, Ronghui Gu, and Jason Nieh. 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.

[50] Jianan Yao, Runzhou Tao, Ronghui Gu, Jason Nieh, Suman Jana, and Gabriel Ryan. DistAI: Data-Driven Automated Invariant Learning for Distributed Protocols. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2021)*, pages 405–421. USENIX Association, July 2021.

[51] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model Checking TLA+ Specifications. In Laurence Pierre and Thomas Kropf, editors, *Correct Hardware Design and Verification Methods*, pages 54–66, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

APPENDIX

A. Proof of Theorem 1

Theorem 1 follows from the four theorems below.

Theorem 2. *Let r be a safety violation of a completion of the sketch S. Then $\pi_{safe}(r)$ is optimal w.r.t. r and S.*

Proof. For brevity, $\pi := \pi_{safe}(r)$. To show that π is optimal, we must show that for any completion X of the sketch S, X satisfies π if and only if r is not a run of X. First suppose X satisfies π. Then π is not an empty disjunction and moreover there exists a $\tau_{A(\vec{v}),i}$ in π that is satisfied by X. This τ-term corresponds to some transition $s \xrightarrow{A(\vec{v})} t$ in r. If the h_i corresponding to the τ-term is a pre-hole, then the action $A(\vec{v})$ is disabled in state s of X. Suppose h_i is a post-hole corresponding to the state variable x. Then x has some value v_x in t. Because X satisfies π, we know that after taking action $A(\vec{v})$ in state s, x has some value $v_x^\star \neq v_x$. In either case, r is not a run of X because it cannot transition from s to t by action $A(\vec{v})$.

Now suppose X does not satisfy π. Then none of the τ-terms in π are satisfied. Let $T = s \xrightarrow{A(\vec{v})} t$ be a transition in r. We will show that X contains all such T and therefore r is a run of X. There are two cases: (1) the action of T has holes in it, or (2) it does not. In case (2), T is a transition that is present in every completion of the sketch. In case (1), we can leverage that X violates all $\tau_{A(\vec{v}),i}$ that were constructed for T. If it violates all $\tau_{A(\vec{v}),i}$ for all pre-holes, then $A(\vec{v})$ is

enabled in state s. If it violates all $\tau_{A(\vec{v}),i}$ for all post-holes, then t is a successor of s by action $A(\vec{v})$ in X. □

Theorem 3. *Let r be a deadlock violation of a completion of the sketch S. Then $\pi_{dead}(r)$ is sub-optimal w.r.t. r and S.*

Proof. Let $\pi := \pi_{dead}(r)$ for brevity. π is under-pruning because although ρ terms ensure that some pre-condition for some action A is weakened for deadlocked state s_k, it is possible that multiple pre-conditions need to be weakened in order for A to be taken in s_k.

Let X be a completion of the sketch S that does not satisfy π. To show that π is not over-pruning, we must show that r is a run of X and that the final state is deadlocked in X. As with the π_{safe} proof, we know that X has all the transitions in r, since all τ-terms are violated. Furthermore, we know that the final state of r is deadlocked in X because all ρ-terms are violated and therefore no pre-condition is weak enough to be taken in order to escape the deadlocked state. □

Theorem 4. *Let r be a liveness violation of a completion of the sketch S. Then $\pi_{live}(r)$ is sub-optimal w.r.t. r and S.*

Proof. Let $\pi := \pi_{live}(r)$. As with the deadlock case, π is under-pruning because X satifying π may only weaken one pre-condition of a fair action where it is necessary to weaken multiple pre-conditions to enable a fair action and make a cycle unfair.

Let X be a completion of the sketch S that does not satisfy π. Then all τ-terms are violated, so r is a run of X, so long as fairness constraints are satisfied. Fairness constraints are satisfied because all of the ρ-terms in π are violated. I.e., ρ-terms ensure there does not exist a fair action in X that is enabled in the cycle of r. □

Theorem 5. *Let r be a stuttering violation of a completion of the sketch S. Then $\pi_{stut}(r)$ is sub-optimal w.r.t. r and S.*

Proof. Let $\pi := \pi_{stut}(r)$ and suppose X_0 is a completion of S that exhibits r. As with the deadlock and liveness violations, π is under-pruning because X satisfying π may only weaken one pre-condition of a fair action where it is necessary to weaken multiple pre-conditions to enable a fair action and make stuttering unfair.

Let X be a completion of the sketch S that does not satisfy π. We must show that r is a run of X. All terms of $\pi_{safe}(r)$ are violated, so the last state, s_k, of r is reachable in X by taking the sequence of transitions in r. Now, each fair action A of X is either enabled or disabled in s_k. We must show that all enabled fair actions are self-looping. Because the terms in π'_ρ are violated, we know that the non-self-looping fair actions that were disabled in state s_k of X_0 are also disabled in state s_k of X. Because the terms in π_τ are violated, we know that states that were self-looping in X_0 are still self-looping in X, if they are enabled in X. □