Synthesis of Distributed Protocols by Enumeration Modulo Isomorphisms

Derek Egolf(✉) and Stavros Tripakis

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

Abstract. Synthesis of distributed protocols is a hard, often undecidable, problem. *Completion* techniques provide partial remedy by turning the problem into a search problem. However, the space of candidate completions is still massive. In this paper, we propose optimization techniques to reduce the size of the search space by a factorial factor by exploiting symmetries (*isomorphisms*) in functionally equivalent solutions. We present both a theoretical analysis of this optimization as well as empirical results that demonstrate its effectiveness in synthesizing both the Alternating Bit Protocol and Two Phase Commit. Our experiments show that the optimized tool achieves a speedup of approximately 2 to 10 times compared to its unoptimized counterpart.

1 Introduction

Distributed protocols are at the heart of the internet, data centers, cloud services, and other types of infrastructure considered indispensable in a modern society. Yet distributed protocols are also notoriously difficult to get right, and have therefore been one of the primary application domains of formal verification [15,19,20,22,31]. An even more attractive proposition is distributed protocol *synthesis*: given a formal correctness specification ψ, automatically generate a distributed protocol that satisfies ψ, i.e., that is *correct-by-construction*.

Synthesis is a hard problem in general, suffering, like formal verification, from scalability and similar issues. Moreover, for distributed systems, synthesis is generally undecidable [12,23,29,30]. Techniques such as program *sketching* [26,27] remedy scalability and undecidability concerns essentially by turning the synthesis problem into a *completion* problem [2,3]: given an *incomplete* system M_0 and a specification ψ, automatically synthesize a completion M of M_0, such that M satisfies ψ.

For example, the synthesis of the well-known *alternating-bit protocol* (ABP) is considered in [4] as a completion problem: given an ABP system containing the incomplete $Sender_0$ and $Receiver_0$ processes shown in Fig. 1, complete these two processes (by adding but not removing any transitions, and not adding nor removing any states), so that the system satisfies a given set of requirements.

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2023
É. André and J. Sun (Eds.): ATVA 2023, LNCS 14215, pp. 270–291, 2023.
https://doi.org/10.1007/978-3-031-45329-8_13

In cases where the space of all possible completions is finite, completion turns synthesis into a decidable problem.[1] However, even then, the number of possible completions can be prohibitively large, even for relatively simple protocols. For instance, as explained in [4], the number of all possible completions in the ABP example is $512^4 \cdot 36$, i.e., approximately 2.5 trillion candidate completions.

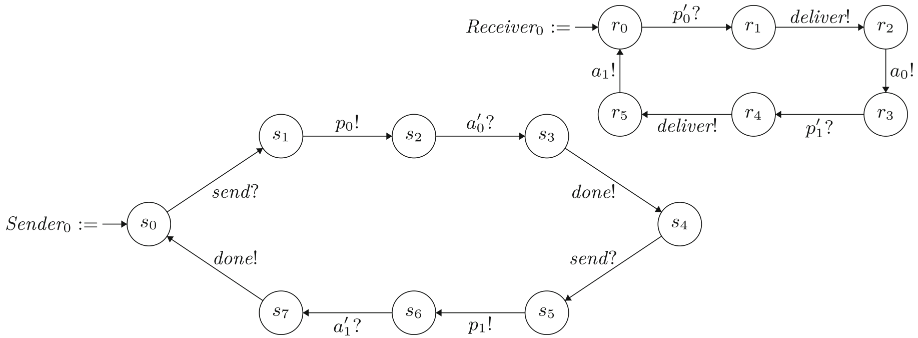

Fig. 1. The incomplete ABP Sender and Receiver processes of [4]

Not only is the number of candidate completions typically huge, but it is often also interesting to generate not just one correct completion, but many. For instance, suppose both M_1 and M_2 are (functionally) correct solutions. We may want to evaluate M_1 and M_2 also for *efficiency* (perhaps using a separate method) [10]. In general, we may want to synthesize (and then evaluate w.r.t. performance or other metrics) not just one, but in principle *all* correct completions. We call this problem the completion *enumeration* problem, which is the main focus of this paper.

Enumeration is harder than *1-completion* (synthesis of just one correct solution), since the number of correct solutions might be very large. For instance, in the case of the ABP example described above, the number of correct completions is 16384 and it takes 88 min to generate all of them [4].

The key idea in this paper is to exploit the notion of *isomorphisms* in order to reduce the number of correct completions, as well as the search space of candidate completions in general. To illustrate the idea, consider a different incomplete $Sender_0$ process, shown in Fig. 2. Two possible completions of this $Sender_0$ are shown in Fig. 3. Although these two completions are in principle different, they are identical except that states s_3 and s_7 are swapped. Our goal is to develop a technique which considers these two completions *equivalent up to isomorphism*, and only explores (and returns) one of them.

[1] We emphasize that no generality is lost in the sense that one can augment the search for correct completions with an outer loop that keeps adding extra *empty* states (with no incoming or outgoing transitions), which the inner completion procedure then tries to complete. Thus, we can keep searching for progressively larger systems (in terms of number of states) until a solution is found, if one exists.

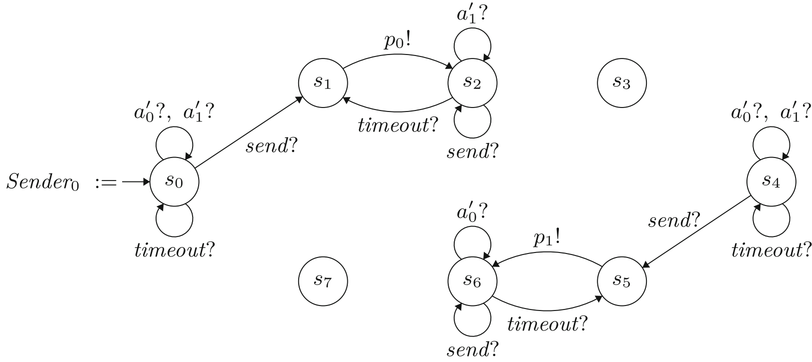

Fig. 2. An incomplete ABP Sender with permutable states s_3, s_7

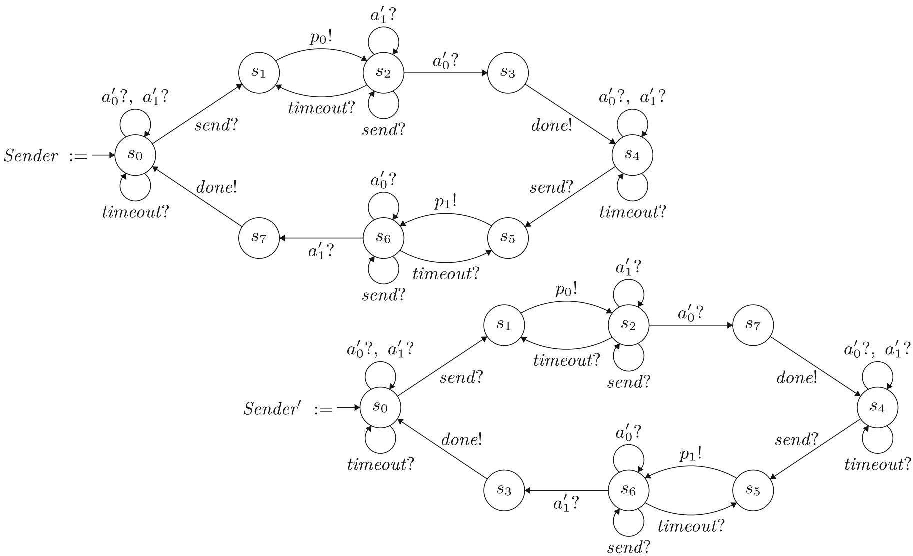

Fig. 3. Two synthesized completions of the incomplete process of Fig. 2. Observe that the two completions are identical except that states s_3 and s_7 are flipped.

To achieve this goal, we adopt the *guess-check-generalize* paradigm (GCG) [1, 2, 13, 26, 27]. In a nutshell, GCG works as follows: (1) pick a candidate completion M; (2) check whether M satisfies ψ: if it does, M is one possible solution to the synthesis problem; (3) if M violates ψ, *prune* the search space of possible completions by excluding a *generalization* of M, and repeat from step (1). In the most trivial case, the generalization of M contains only M itself. Ideally, however, and in order to achieve a more significant pruning of the search space, the generalization of M should contain many more "bad" completions which are somehow "similar" (for instance, isomorphic) to M.

A naive way to generalize based on isomorphism is to keep a list of completions encountered thus far and perform an isomorphism check against every element of this list whenever a new candidate is picked. Our approach is smarter: in fact, it does not involve any isomorphism checks whatsoever. Instead, our approach guarantees that no isomorphic completions are ever picked to begin with by pruning them from the search space. This is ultimately done using syntactic transformations of completion representations. The details are left for Sect. 4.

Furthermore, our notion of "encountering" a completion is quite wide. Rather than just pruning completions that are isomorphic to *candidates*, we also prune completions that are isomorphic to any completion in the *generalizations of* the candidates (with respect to some prior, unextended notion of generalization). Between the trivial approach involving isomorphism checks and our own approach are several other approaches which are good, but not excellent. Indeed, a categorization of the subtle differences between such approaches is a key contribution of this paper (see Sect. 4.3). These subtleties are easy to miss.

In summary, the main contributions of this paper are the following: (1) we define the 1-completion and completion-enumeration problems *modulo isomorphisms*; (2) we examine new methods to solve these problems based on the GCG paradigm; (3) we identify properties that an efficient GCG modulo isomorphisms algorithm should have; (4) we propose two instances of such an algorithm, using a naive and a sophisticated notion of generalization; (5) we evaluate our methods on the synthesis of two simple distributed protocols: the ABP and Two Phase Commit (2PC) and demonstrate speedups with respect to the unoptimized method of approximately 2 to 10 times.

2 Preliminaries

Labeled Transition Systems. A (finite) *labeled transition system* (LTS) M is a tuple $\langle \Sigma, Q, Q_0, \Delta \rangle$, where

- Σ is a finite set of transition *labels*
- Q is a finite set of *states*
- $Q_0 \subseteq Q$ is the set of *initial states*
- $\Delta \subseteq Q \times \Sigma \times Q$ is the *transition relation*.

We write the transition $(p, a, q) \in \Delta$ as $p \xrightarrow{a} q$.

A *run* of M is an infinite sequence $q_0 \xrightarrow{a_0} q_1 \xrightarrow{a_1} q_2 \xrightarrow{a_2} ...$, where $q_0 \in Q_0$ and for each i we have $(q_i, a_i, q_{i+1}) \in \Delta$. The *trace* produced by this run is $a_0 a_1 a_2 \cdots$. Semantically, an LTS M represents a set of infinite traces, denoted $[\![M]\!] \subseteq \Sigma^\omega$. Specifically, a trace $a_0 a_1 a_2 \cdots$ is in $[\![M]\!]$ exactly when there exists a run $q_0 \xrightarrow{a_0} q_1 \xrightarrow{a_1} q_2 \xrightarrow{a_2} ...$ of M.

Correctness Specification. We will assume that we have some formal notion of *specification* and some formal notion of *satisfaction* between an LTS M and a specification ψ. We write $M \vDash \psi$ to denote that M satisfies ψ. Our work is agnostic to what exactly ψ might be (e.g., a temporal logic formula, etc.).

Completions and Syntactic Constraints. Suppose that M and M_0 are two LTSs with the same set of labels Σ, the same set of states Q, the same set of initial states Q_0, and with transition relations Δ and Δ_0, respectively. We say that M *is a completion of* M_0 exactly when $\Delta_0 \subseteq \Delta$. That is, M completes M_0 by adding more transitions to it (and not removing any). For example, each of the two LTSs of Fig. 3 is a completion of the LTS shown in Fig. 2.

Often, we wish to impose some constraints on the kind of synthesized processes that we want to obtain during automated synthesis, other than the global constraints imposed on the system by the correctness specification. For example, in the formal distributed protocol model proposed in [4], synthesized processes such as the ABP *Sender* and *Receiver* are constrained to satisfy a number of requirements, including absence of deadlocks, determinism of the transition relation, the constraint that each state is either an *input state* (i.e., it only receives inputs) or an *output state* (i.e., it emits a unique output), the constraint that input states are *input-enabled* (i.e., they do not block any inputs), and so on. Such properties are often syntactic or structural and can be inferred statically by observing the transition relation. The fact that an LTS is a completion of another LTS can also be captured by such constraints.

Constraints like the above are application-specific, and our approach is agnostic to their precise form and meaning. We will therefore abstract them away, and assume that there is a propositional logic formula Φ which captures the set of all syntactically well-formed candidate completions. The variable space of Φ and its precise meaning is application-specific. We will give a detailed construction of Φ for LTS in Sect. 3. We write $M \vDash \Phi$ when LTS M satisfies the *syntactic constraints* Φ. Let $[\![\Phi]\!] = \{M \mid M \vDash \Phi\}$.

We say that an LTS is *correct* if it satisfies both the syntactic constraints imposed by Φ and the semantic constraints imposed by ψ.

Computational Problems

Problem 1 (Model-Checking). Given LTS M, specification ψ, and constraints Φ, check whether $M \vDash \psi$ and $M \vDash \Phi$.

A solution to the model-checking problem is an algorithm, MC, such that for all M, Φ, ψ, if $M \vDash \Phi$ and $M \vDash \psi$ then $\text{MC}(M, \Phi, \psi) = 1$; otherwise, $\text{MC}(M, \Phi, \psi) = 0$.

Problem 2 (Synthesis). Given specification ψ and constraints Φ, find, if one exists, LTS M such that $M \vDash \psi$ and $M \vDash \Phi$.

Problem 3 (Completion). Given LTS M_0, specification ψ, and constraints Φ, find, if one exists, a completion M of M_0 such that $M \vDash \psi$ and $M \vDash \Phi$.

Problem 4 (Completion enumeration). Given LTS M_0, specification ψ, and constraints Φ, find all completions M of M_0 such that $M \vDash \psi$ and $M \vDash \Phi$.

3 The Guess-Check-Generalize Paradigm

In this section we first propose a generic GCG algorithm and reason about its correctness (Sect. 3.1). We then show how to instantiate this algorithm to solve Problems 3 and 4 (Sect. 3.2).

3.1 A Generic GCG Algorithm and Its Correctness

Algorithm 1 is a formal description of a generic GCG algorithm. The algorithm takes as input: (1) a set of syntactic constraints in the form of a propositional formula Φ, as described in Sect. 2; (2) a specification ψ as described in Sect. 2; and (3) a *generalizer* function γ, described below.

Algorithm 1: GCG$[\Phi, \psi, \gamma]$

```
1 while Φ is satisfiable do
2     σ := SAT(Φ);
3     if MC(M_σ, Φ, ψ) = 1 then
4         return σ;
5         Φ := Φ ∧ ¬σ;
6     else
7         Φ := Φ ∧ ¬γ(σ);
```

Φ is a propositional logic formula (over a certain set of boolean variables that depends on the application domain at hand) encoding all possible syntactically valid completions. Every satisfying assignment σ of Φ corresponds to one completion, which we denote as M_σ. Observe that GCG does not explicitly take an initial (incomplete) model M_0 as input: this omission is not a problem because M_0 can be encoded in Φ, as mentioned in Sect. 2. We explain specifically how to do that in the case of LTS in Sect. 3.2.

The algorithm works as follows: while Φ is satisfiable: Line 2: pick a candidate completion σ allowed by Φ by calling a SAT solver. Line 3: model-check the corresponding model M_σ against ψ (by definition, M_σ satisfies Φ because σ satisfies Φ). Line 4: if M_σ satisfies ψ then we have found a correct model: we can return it and terminate if we are solving Problem 3, or return it and continue our search for additional correct models if we are solving Problem 4. In the latter case, in line 5 we exclude σ from Φ (slightly abusing notation, we treat σ as a formula satisfied exactly and only by σ, so that $\neg\sigma$ is the formula satisfied by all assignments except σ). Line 7: if M_σ violates ψ, then we exclude from Φ the *generalization* $\gamma(\sigma)$ of σ, and continue our search.

Generalizers. A *generalizer* is a function γ which takes an assignment σ and returns a propositional logic formula $\gamma(\sigma)$ that encodes all "bad" assignments that we wish to exclude from Φ. Ideally, however, $\gamma(\sigma)$ will encode many more assignments (and therefore candidate completions), so as to prune as large a

part of the search space as possible. A concrete implementation of γ may require additional information other than just σ. For example, γ may consult the specification ψ, counter-examples returned by the model-checker (which are themselves a function of ψ and σ), and so on. We avoid including all this information in the inputs of γ to ease presentation. We note that ψ does not change during a run of Algorithm 1 and therefore ψ can be "hardwired" into γ without loss of generality.

A valid generalizer should include the assignment being generalized and it should only include bad assignments (i.e., it should exclude correct completions). Formally, a generalizer γ is said to be *proper* if for all σ such that $\sigma \vDash \Phi$ and $M_\sigma \nvDash \psi$, the following conditions hold: (1) *Self-inclusion*: $\sigma \vDash \gamma(\sigma)$, and (2) *Correct-exclusion*: for any ϱ, if $\varrho \vDash \Phi$ and $M_\varrho \vDash \psi$ then $\varrho \nvDash \gamma(\sigma)$.

The Correctness of GCG

Lemma 1. *If γ is proper then* GCG$[\Phi, \psi, \gamma]$ *terminates.*

Proof. If γ is proper then $\gamma(\sigma)$ is guaranteed to include at least σ. Φ is a propositional logic formula, therefore it only has a finite set of satisfying assignments. Every iteration of the loop removes at least one satisfying assignment from Φ, therefore the algorithm terminates. □

During a run, Algorithm 1 returns a (possibly empty) set of assignments $\mathsf{Sol} = \{\sigma_1, \sigma_2, ..., \sigma_n\}$, representing the solution to Problems 3 or 4. Also during a run, the algorithm guesses candidate assignments by calling the subroutine SAT (line 2). Let Cand be the set of all these candidates. Note that $\mathsf{Sol} \subseteq \mathsf{Cand}$, since every solution returned (line 4) has been first guessed in line 2.

Whenever the algorithm reassigns $\Phi := \Phi \land \neg\varphi$, we say that it *prunes* φ, i.e., the satisfying assignments of φ are now excluded from the search. We will need to reason about the set of assignments that have been pruned after a certain *partial run* of the program. In such cases we can imagine running the algorithm for some amount of time and pausing it. Then the set Pruned denotes the set of assignments that have been pruned up until that point. It is true that after the program terminates $\mathsf{Pruned} = [\![\Phi]\!] \backslash \mathsf{Cand}$, but this equality does not necessarily hold for all partial runs.

Theorem 1. *(1)* GCG$[\Phi, \psi, \gamma]$ *is* sound*, i.e., for all $\sigma \in \mathsf{Sol}$, we have $\sigma \vDash \Phi$ and $M_\sigma \vDash \psi$. (2) If γ is proper then* GCG$[\Phi, \psi, \gamma]$ *is* complete*, i.e., for all $\sigma \vDash \Phi$, if $M_\sigma \vDash \psi$ then $\sigma \in \mathsf{Sol}$.*

Proof. Every $\sigma \in \mathsf{Sol}$ satisfies Φ (line 2) and the corresponding M_σ satisfies ψ (line 3), therefore GCG$[\Phi, \psi, \gamma]$ is sound. Now, suppose that γ is proper, and take ϱ such that $\varrho \vDash \Phi$ and $M_\varrho \vDash \psi$. To show completeness, it suffices to show that $\varrho \in \mathsf{Cand}$. Then, we also have $\varrho \in \mathsf{Sol}$ because M_ϱ passes the model-checking test in line 3. Suppose, for a contradiction, that $\varrho \notin \mathsf{Cand}$, i.e., that ϱ is pruned. Then there must exist some σ such that $\varrho \vDash \gamma(\sigma)$ (line 7). But $\sigma \vDash \Phi$ (line 2), which means that ϱ violates the *correct-exclusion* property of γ. Contradiction. □

3.2 A Concrete Instance of GCG for LTS

Algorithm 1 is *generic* in the sense that depending on how exactly we instantiate Φ, ψ, and γ, we can encode different completion enumeration (and more generally model enumeration) problems, as well as solutions. We now show how to instantiate Algorithm 1 to solve Problems 3 and 4 concretely for LTS.

Encoding LTSs and Completions in Propositional Logic. Let $M_0 = \langle \Sigma, Q, Q_0, \Delta_0 \rangle$ be an incomplete LTS. Then we can define a set of boolean variables

$$V := \{p \looparrowright^a q \mid p, q \in Q \wedge a \in \Sigma\}$$

so that boolean variable $p \looparrowright^a q$ encodes whether transition $p \xrightarrow{a} q$ is present or not (if $p \xrightarrow{a} q$ is present, then $p \looparrowright^a q$ is true, otherwise it is false). More formally, let $\textsc{asgn}_V$ be the set of all assignments over V. An assignment $\sigma \in \textsc{asgn}_V$ represents LTS M_σ with transition relation $\Delta_\sigma = \{(p, a, q) \mid \sigma(p \looparrowright^a q) = 1\}$. To enforce M_σ to be a completion of M_0, we need to enforce that $\Delta_0 \subseteq \Delta_\sigma$. We do so by initializing our syntactic constraints Φ as $\Phi := \Phi_{\Delta_0}$, where

$$\Phi_{\Delta_0} := \bigwedge_{p \xrightarrow{a} q \in \Delta_0} p \looparrowright^a q.$$

We can then add extra constraints to Φ such as determinism or absence of deadlocks, as appropriate.

A Concrete Generalizer for LTS. Based on the principles of [4], we can construct a *concrete generalizer* $\gamma_{LTS}(\sigma)$ for LTS as $\gamma_{LTS}(\sigma) := \gamma_{safe}(\sigma) \vee \gamma_{live}(\sigma)$, which we separate into a disjunction of a safety violation generalizer and a liveness violation generalizer. The safety component γ_{safe} works on the principle that if LTS M_σ violates a safety property, then adding extra transitions will not solve this violation. Thus:

$$\gamma_{safe}(\sigma) := \bigwedge_{\{x \in V \mid \sigma(x) = 1\}} x.$$

The liveness component γ_{live} can be defined based on a notion of reachable, “bad” cycles that enable something to happen infinitely often. Thus, $\neg\gamma_{live}$ captures all LTSs that disable these bad cycles by breaking them or making them unreachable.

It can be shown that the concrete generalizer γ_{LTS} is proper. Therefore, the concrete instance $\textsc{gcg}[\Phi, \psi, \gamma_{LTS}]$ is sound, terminating, and complete, i.e., it solves Problems 3 and 4.

Even though the concrete generalizer is correct, it is not very effective. In particular, it does not immediately prune isomorphisms. There may be $O(n!)$ trivially equivalent completions up to state reordering, where n is the number of states in the LTS. In the next section we present two optimizations exploiting isomorphisms.

4 Synthesis Modulo Isomorphisms

4.1 LTS Isomorphisms

Intuitively, two LTS are isomorphic if we can rearrange the states of one to obtain the other. For synthesis purposes, we often wish to provide as a constraint a set of *permutable states* A, so as to exclude rearrangements that move states outside of A. If we can still rearrange the states of an LTS M_1 to obtain another LTS M_2 subject to this constraint, then we say that M_1 *and* M_2 *are isomorphic up to* A. For example, the two LTSs of Fig. 3 are isomorphic up to the set of permutable states $A = \{s_3, s_7\}$. Strictly speaking, they are permutable up to any set of their states, but we choose A to reflect the fact that those two states have no incoming or outgoing transitions in Fig. 2. Permuting any other states would yield an LTS that is not a completion of Fig. 2.

We now define isomorphisms formally. Let M_0, M_1, and M_2 be LTSs with the same Σ, Q, Q_0, and with transition relations Δ_0, Δ_1, and Δ_2, respectively. Suppose that M_1 and M_2 are both completions of M_0. Let $A \subseteq Q \backslash Q_0$. Then we say M_1 and M_2 are isomorphic up to A, denoted $M_1 \overset{A}{\simeq} M_2$, if and only if there exists a bijection $f : A \to A$ (i.e., a *permutation*) such that

$$p \xrightarrow{a} q \in \Delta_1 \text{ if and only if } f(p) \xrightarrow{a} f(q) \in \Delta_2.$$

By default, we will assume that A is the set of non-initial states that have no incoming or outgoing transitions in M_0. In that case we will omit A and write $M_1 \simeq M_2$.

Lemma 2. *LTS isomorphism is an equivalence relation, i.e., it is reflexive, symmetric, and transitive.*

We use $[M]$ to denote the *equivalence class* of M, i.e., $[M] = \{M' \mid M' \simeq M\}$.

Lemma 3. *If* $M_1 \overset{A}{\simeq} M_2$ *then* $[\![M_1]\!] = [\![M_2]\!]$.

Lemma 3 states that LTS isomorphism preserves traces. More generally, we will assume that our notion of specification is preserved by LTS isomorphism, namely, that if $M_1 \overset{A}{\simeq} M_2$ then for any specification ψ, $M_1 \models \psi$ iff $M_2 \models \psi$.

Isomorphic Assignments. Two assignments σ and ϱ are isomorphic if the LTSs that they represent are isomorphic. Hence we write $\sigma \simeq \varrho$ if and only if $M_\sigma \simeq M_\varrho$. We write $[\varrho]$ to denote the equivalence class of ϱ, i.e., the set of all assignments that are isomorphic to ϱ. These equivalence classes partition Φ since $\simeq$ is an equivalence relation.

4.2 Completion Enumeration Modulo Isomorphisms

Isomorphisms allow us to focus our attention to Problem 5 instead of Problem 4:

Problem 5 (Completion enumeration modulo isomorphisms). Given LTS M_0, specification ψ, and constraints Φ, find the set

$$\{[M] \mid M \text{ is a completion of } M_0 \text{ such that } M \vDash \psi \text{ and } M \vDash \Phi\}.$$

Problem 5 asks that only significantly different (i.e., non-isomorphic) completions are returned to the user. Problem 5 can be solved by a simple modification to Algorithm 1, namely, to exclude the entire equivalence class $[\sigma]$ of any discovered solution σ, as shown in Algorithm 2, line 5.

Algorithm 2: $\text{GCG}_{\simeq}[\Phi, \psi, \gamma]$ solving Problem 5

```
1 while Φ is satisfiable do
2     σ := SAT(Φ);
3     if MC(M_σ, Φ, ψ) = 1 then
4         return σ;
5         Φ := Φ ∧ ¬[σ];
6     else
7         Φ := Φ ∧ ¬γ(σ);
```

4.3 Properties of an Efficient GCG Algorithm

We begin by presenting a list of properties that an efficient instance of GCG ought to satisfy. Except for Property 1, satisfaction of these properties generally depends on the generalizer used.

Property 1. For all σ that satisfy Φ, $[\sigma] \cap \mathsf{Sol}$ has 0 or 1 element(s). In other words, we return at most one solution per equivalence class.

Property 1 asks that only significantly different (i.e., non-isomorphic) completions are returned to the user, thereby solving Problem 5, which is our main goal. In addition, this property implies that the number of completions is kept small, which is important when these are fed as inputs to some other routine (e.g., one that selects a "highly fit" completion among all valid completions).

$\text{GCG}_{\simeq}$ satisfies Property 1, regardless of the parameters. However, we can go further, by ensuring that not only we do not return isomorphic completions, but we do not even consider isomorphic candidate completions in the first place:

Property 2. For all σ that satisfy Φ, $[\sigma] \cap \mathsf{Cand}$ has 0 or 1 element(s). In other words, we consider at most one candidate per equivalence class.

Maintaining Property 2 now guarantees that we only call the most expensive subroutines at most once for each equivalence class. Note that, since $\mathsf{Sol} \subseteq \mathsf{Cand}$, Property 2 implies Property 1.

Property 2 is still not entirely satisfactory. For instance, suppose the algorithm generates σ as a candidate and then prunes $\gamma(\sigma)$. Now suppose that $\varrho \simeq \sigma$. Property 2 implies that we *cannot* call/prune $\gamma(\varrho)$. Property 3 rectifies this:

Property 3 (invariant). Suppose that $\text{GCG}_{\simeq}$ invokes $\Phi := \Phi \wedge \neg\gamma(\sigma)$. Then for any $\varrho \simeq \sigma$, we should have $[\![\gamma(\varrho)]\!] \subseteq \mathsf{Pruned}$. In other words, if we prune $\gamma(\sigma)$, we should also prune $\gamma(\varrho)$ for every ϱ isomorphic to σ.

We note that, contrary to Properties 1 and 2 which need only hold after termination, Property 3 is an *invariant*: we want it to hold for all *partial executions* of the algorithm.

Theorem 2. *Suppose γ is proper. If $\text{GCG}_{\simeq}[\Phi, \psi, \gamma]$ maintains Property 3 as an invariant, then $\text{GCG}_{\simeq}[\Phi, \psi, \gamma]$ also maintains Property 2.*

Maintaining Property 3 increases the rate at which the search space is pruned, but is still not enough. Suppose that $\tau \vDash \gamma(\sigma)$ and that $\tau' \simeq \tau$. If we prune the members of $\gamma(\sigma)$, then we will prune τ, but not necessarily τ'. This possibility is unsatisfactory, since τ and τ' should both be treated whenever one of them is.

Property 4 (invariant). Suppose $\tau \in \mathsf{Pruned}$ and $\tau' \simeq \tau$. Then $\tau' \in \mathsf{Pruned}$ or $\tau' \in \mathsf{Sol}$. In other words, if we prune τ we should also prune any isomorphic τ', unless τ' happens to be a solution. (Note that Property 1 guarantees that this exception applies to at most one τ').

Maintaining Property 4 as an invariant further accelerates pruning. Under certain conditions, Property 3 implies Property 4. In particular, Property 3 implies Property 4 if γ is *invertible*, a concept that we define next.

Invertible Generalizers. Let γ be a generalizer and let τ be an assignment. We define the *inverse* $\gamma^{-1}(\tau)$, to be the propositional logic formula satisfied by all σ such that $\tau \vDash \gamma(\sigma)$. That is, $\sigma \vDash \gamma^{-1}(\tau)$ iff $\tau \vDash \gamma(\sigma)$.

Let φ and φ' be propositional logic formulas. Suppose that for every $\sigma \vDash \varphi$, there exists a $\sigma' \vDash \varphi'$ such that $\sigma' \simeq \sigma$. Then we say that φ *subsumes* φ' *up to isomorphism*. If φ and φ' both subsume each other, then we say that they are *equivalent up to isomorphism*.

A generalizer γ is *invertible* if for all assignments τ, τ' that satisfy Φ, if $\tau \simeq \tau'$ then $\gamma^{-1}(\tau)$ and $\gamma^{-1}(\tau')$ are equivalent up to isomorphism. Now if $\tau \vDash \gamma(\sigma)$ and $\tau' \simeq \tau$, invertibility guarantees that we can point to a $\sigma' \simeq \sigma$ such that $\tau' \vDash \gamma(\sigma')$.

Theorem 3. *Suppose γ is proper and invertible. If $\text{GCG}_{\simeq}[\Phi, \psi, \gamma]$ maintains Property 3 as an invariant, then $\text{GCG}_{\simeq}[\Phi, \psi, \gamma]$ also maintains Property 4 as an invariant.*

Proof. Let γ be a proper, invertible generalizer. We will proceed by contradiction. Assume that we have run the algorithm for some amount of time and paused its execution, freezing the state of Pruned. Suppose that $\text{GCG}_{\simeq}[\Phi, \psi, \gamma]$ satisfies Property 3 at this point, but that it does not satisfy Property 4. From the negation of Property 4, we have at this point in the execution two assignments τ and τ' such that (1) $\tau \simeq \tau'$, (2) $\tau \in \mathsf{Pruned}$, (3) $\tau' \notin \mathsf{Pruned}$, and (4) $\tau' \notin \mathsf{Sol}$.

There are two cases that fall out of (2). Either τ was pruned using a call to γ, or exactly $[\tau]$ was pruned. In the second case, we quickly reach a contradiction since it implies that $\tau' \in \mathsf{Pruned}$, violating assumption (3).

So instead, we assume $\tau \models \gamma(\sigma)$ for some σ and that this call to γ was invoked at some point in the past. So $\sigma \models \gamma^{-1}(\tau)$. But then by invertibility and (1) there exists $\sigma' \simeq \sigma$ such that $\sigma' \models \gamma^{-1}(\tau')$ and hence $\tau' \models \gamma(\sigma')$. Property 3 tells us then that $\tau' \in \mathsf{Pruned}$, but this conclusion also violates assumption (3). □

It can be shown that the generalizer γ_{LTS} is invertible. Essentially, this is because γ_{LTS} does not depend on state names (for example, the structure of cycles and paths is independent of state names). Still, $\text{GCG}_{\simeq}[\Phi, \psi, \gamma_{LTS}]$ satisfies only Property 1 above. Therefore, we will next describe an optimized generalization method that exploits isomorphism to satisfy all properties.

4.4 Optimized Generalization

Equivalence Closure. If γ is a generalizer and $\simeq$ is an equivalence relation, then let

$$\overset{\simeq}{\gamma}(\varrho) := \bigvee_{\sigma \in [\varrho]} \gamma(\sigma)$$

be the *equivalence closure* of γ. If $\gamma(\sigma) \equiv \overset{\simeq}{\gamma}(\sigma)$ for all σ, we say that γ is *closed under equivalence.*

Note that $\overset{\simeq}{\gamma}$ is itself a generalizer. An instance of $\text{GCG}_{\simeq}$ that uses $\overset{\simeq}{\gamma}$ is correct and satisfies all the efficiency properties identified above:

Theorem 4. *If γ is a proper generalizer, then $\text{GCG}_{\simeq}[\Phi, \psi, \overset{\simeq}{\gamma}]$ is sound, terminating, and complete up to isomorphisms.*

Theorem 5. *If γ is proper, then $\text{GCG}_{\simeq}[\Phi, \psi, \overset{\simeq}{\gamma}]$ maintains Properties 1 and 2. Furthermore, the algorithm maintains Property 3 as an invariant.*

Theorem 6. *If γ is both proper and invertible, then: (1) $\overset{\simeq}{\gamma}$ is invertible; (2) $\text{GCG}_{\simeq}[\Phi, \psi, \overset{\simeq}{\gamma}]$ maintains Property 4 as an invariant.*

Computation Options for $\overset{\simeq}{\gamma}$. The naive way to compute $\overset{\simeq}{\gamma}$ is to iterate over all $\sigma_1, \sigma_2, \cdots, \sigma_k \in [\varrho]$, compute each $\gamma(\sigma_i)$, and then return the disjunction of all $\gamma(\sigma_i)$. We call this the *naive generalization* approach. The problem with this approach is that we have to call γ as many as $n!$ times, where n is the number of permutable states. The experimental results in Sect. 5 indicate empirically that this naive method does not scale well.

We thus propose a better approach, which is *incremental*, in the sense that we only have to compute γ once, for $\gamma(\sigma_1)$; we can then perform simple *syntactic transformations* on $\gamma(\sigma_1)$ to obtain $\gamma(\sigma_2)$, $\gamma(\sigma_3)$, and so on. As we will show, these transformations are much more efficient than computing each $\gamma(\sigma_i)$ from scratch. So-called *permuters* formalize this idea:

Permuters. A *permuter* is a function π that takes as input an assignment ϱ and the generalization $\gamma(\sigma)$ for some $\sigma \simeq \varrho$, and returns a propositional logic formula $\pi(\sigma, \gamma(\varrho))$ such that $\forall \varrho \models \Phi, \forall \sigma \simeq \varrho :: M_\varrho \not\models \psi \rightarrow \pi(\varrho, \gamma(\sigma)) \equiv \gamma(\varrho)$. That is, assuming ϱ is "bad" ($M_\varrho \not\models \psi$), $\pi(\varrho, \gamma(\sigma))$ is equivalent to $\gamma(\varrho)$. However, contrary to γ, π can use the extra information $\gamma(\sigma)$ to compute the generalization of ϱ. Then, instead of $\tilde{\tilde{\gamma}}(\varrho)$, we can compute the logically equivalent formula

$$\gamma_\pi(\varrho) := \bigvee_{\sigma \in [\varrho]} \pi(\sigma, \gamma(\varrho)).$$

Theorem 7. *Theorems 4, 5, and 6 also hold for* $\text{GCG}_\simeq[\Phi, \psi, \gamma_\pi]$.

Proof. Follows from the fact that for any ϱ, $\gamma_\pi(\varrho) \equiv \tilde{\tilde{\gamma}}(\varrho)$. ☐

A Concrete Permuter for LTS. We now explain how to compute π concretely in our application domain, namely LTS. Let M_0 be an incomplete LTS. Let σ_1, σ_2 be two assignments encoding completions M_{σ_1} and M_{σ_2} of M_0. Suppose $M_{\sigma_1} \overset{A}{\simeq} M_{\sigma_2}$. Recall that A is the set of permutable states (the non-initial states with no incoming/outgoing transitions by default). Then there is a permutation $f : A \rightarrow A$, such that applying f to the states of M_{σ_1} yields M_{σ_2}. f allows us to transform one LTS to another, but it also allows us to transform the generalization formula for σ_1, namely $\gamma(\sigma_1)$, to the one for σ_2, namely $\gamma(\sigma_2)$.

For example, let M_0 be the leftmost LTS in Fig. 4, with alphabet $\Sigma = \{a\}$, states $Q = \{p_0, p_1, p_2, p_3\}$, initial state p_0, and the empty transition relation. Let M_{σ_1} and M_{σ_2} be the remaining LTSs shown in Fig. 4. Let $A = \{p_1, p_2, p_3\}$ and let f be the permutation mapping p_1 to p_3, p_3 to p_2, and p_2 to p_1. Then $M_{\sigma_1} \overset{A}{\simeq} M_{\sigma_2}$ and f is the witness to this isomorphism.

Let $\gamma(\sigma_1) = (p_0 \not\rightarrow^a p_1) \wedge (p_1 \not\rightarrow^a p_2) \wedge (p_2 \not\rightarrow^a p_3)$. $\gamma(\sigma_1)$ captures the four LTSs in Fig. 5. The key idea is that we can compute $\gamma(\sigma_2)$ by transforming $\gamma(\sigma_1)$ *purely syntactically*. In particular, we apply the permutation f to all p_i appearing in the variables of the formula. Doing so, we obtain $\gamma(\sigma_2) = (p_0 \not\rightarrow^a p_3) \wedge (p_3 \not\rightarrow^a p_1) \wedge (p_1 \not\rightarrow^a p_2)$. This formula in turn captures the four LTSs in Fig. 6, which are exactly the permutations of those in Fig. 5 after applying f.

We now describe this transformation formally. Observe that M_{σ_1} and M_{σ_2} have the same set of states, say Q. We extend the permutation to $f : Q \rightarrow Q$ by defining $f(q) = q$ for all states $q \notin A$. Now, we extend this permutation of states to permutations of the set V (the set of boolean variables encoding transitions). Specifically we extend f to permute V by defining: $f(p \not\rightarrow^a q) := f(p) \not\rightarrow^a f(q)$ and we extend it to propositional formulas by applying it to all variables in the formula. Then we define $\pi_{LTS}(\sigma_2, \gamma(\sigma_1)) := f(\gamma(\sigma_1))$.

In essence, the permuter π_{LTS} identifies the permutation f witnessing the fact that $\sigma_1 \simeq \sigma_2$. It then applies f to the variables of $\gamma(\sigma_1)$. Applying f to $\gamma(\sigma_1)$ is equivalent to applying f to all assignments that satisfy $\gamma(\sigma_1)$.

It can be shown that π_{LTS} is a permuter for LTS. It follows then that the concrete instance $\text{GCG}_\simeq[\Phi, \psi, \gamma_\pi]$ (where $\gamma := \gamma_{LTS}$ and $\pi := \pi_{LTS}$) satisfies

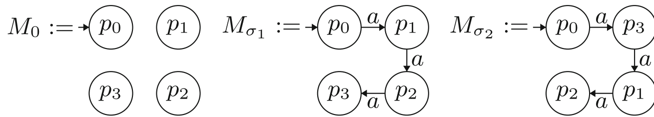

Fig. 4. An incomplete LTS M_0 and two possible completions, M_{σ_1} and M_{σ_2}

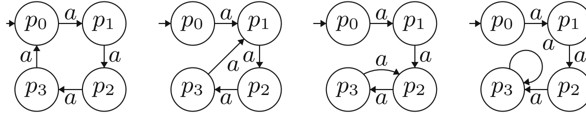

Fig. 5. LTSs represented by $(p_0 \looparrowright^a p_1) \wedge (p_1 \looparrowright^a p_2) \wedge (p_2 \looparrowright^a p_3)$

Theorem 7, i.e., it is sound, terminating, complete up to isomorphisms, and satisfies all Properties 1–4.

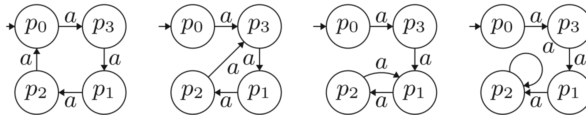

Fig. 6. LTSs represented by $(p_0 \looparrowright^a p_3) \wedge (p_3 \looparrowright^a p_1) \wedge (p_1 \looparrowright^a p_2)$

5 Implementation and Evaluation

Implementation and Experimental Setup. We evaluate the three algorithms discussed so far: the *unoptimized* algorithm $\text{GCG}[\Phi, \psi, \gamma_{LTS}]$ of [2,4] (Sect. 3.2); and the *naive optimization* $\text{GCG}_{\simeq}[\Phi, \psi, \tilde{\tilde{\gamma}}]$ and *permuter optimization* $\text{GCG}_{\simeq}[\Phi, \psi, \gamma_\pi]$ algorithms of Sect. 4.4. These are respectively labeled 'unopt.', 'naive opt.', and 'perm. opt.' in the tables that follow.

In addition, we evaluate the unoptimized algorithm outfitted with an additional optimization, which we call the dead transition optimization. We say that a transition of an LTS is *dead* if this transition is never taken in any run. If M with states Q is correct and has k dead transitions, then there are $|Q|^k$ solutions that are equivalent modulo dead transitions, since we can point a dead transition anywhere while maintaining correctness. The dead transition optimization prunes all solutions which are equivalent modulo dead transitions. It is equivalent to the unoptimized algorithm in cases where there are no solutions or where we are looking for only one solution. Therefore, we evaluate the dead transition optimization side-by-side with the unoptimized solution only when we are enumerating all correct completions. The naive and permuter optimizations both include the dead transition optimization.

We use [28], the Python implementation of $\textsc{gcg}[\Phi, \psi, \gamma_{LTS}]$ made publicly available by the authors of [2,4], and we implement our optimizations on top of [28] in order to keep the comparison fair. The tool can handle completion of distributed systems, rather than of single LTSs. Distributed systems are represented as networks of communicating LTSs similar to those in [4]. Specifications are represented using safety and liveness (Büchi) monitors, again similar to those in [4]. However, let us again mention that our approach is not specific to any particular specification logic; it should allow for performance gains whenever the cost of model-checking is greater than the cost of the simple syntactic transformations applied by the permuter. We use the SAT solver Z3 [7] to pick candidates from the search space. Our experimental results can be reproduced using a publicly available artifact [9].

For our experiments we use the ABP case study as presented in [4] as well as our own two phase commit (2PC) case study. We consider three use cases: (1) *completion enumeration*: enumerate all correct completions; (2) *realizable 1-completion*: return the first correct completion and stop, where we ensure that a correct completion exists; and (3) *unrealizable 1-completion*: return the first correct completion, except that we ensure that none exists (and therefore the tool has to explore the entire candidate space in vain).

We consider a *many-process synthesis* scenario, where the goal is to synthesize two or more processes, and a *1-process synthesis* scenario, where the goal is to synthesize a single process. In both of these scenarios across both the ABP and 2PC case studies, the synthesized processes are composed with additional environment processes and safety and liveness monitors. The results of the many-process synthesis scenario are presented shortly. Due to lack of space, the results of the 1-process synthesis scenario are presented in Appendix A.2 of [11]. The latter results do not add much additional insight, except that 1-process synthesis tends to take less time.

Each experiment was run on a dedicated 2.40 GHz CPU core located on the Northeastern Discovery Cluster. All times are in seconds, rounded up to the second.

Many-Process Synthesis Experiments. In all these experiments, there are multiple LTSs that must both be completed. In the case of ABP: (1) the incomplete ABP $Receiver_0$ of Fig. 1 without further modification; (2) an incomplete sender process, which is obtained by removing some set of transitions from process *Sender* of Fig. 3. The set of transitions removed from *Sender* are all incoming and all outgoing transitions from all states designated as permutable for that experiment (column A in the tables that follow). For instance, in experiment $\{s_1, s_2\}$ of Table 1 we remove all incoming and outgoing transitions from states s_1 and s_2 of *Sender*, and similarly for the other experiments. And in the case of 2PC: (1) two incomplete 2PC database managers (see Fig. 8 in Appendix A.1 of [11]) (2) an incomplete transaction manager, which is obtained by removing some set of transitions from a complete transaction manager (see Fig. 7 in Appendix A.1 of [11]).

Table 1. Many-Process Synthesis, Completion Enumeration

Case Study; A	unopt.			dead opt.			naive opt.			perm. opt.		
	sol.	iter.	time	sol.	iter.	time	sol.	iter.	time	sol.	iter.	time
2PC; $\{p_1, p_2\}$	4	536	47	4	536	46	2	274	34	2	274	28
2PC; $\{p_2, p_3\}$	48	1417	130	4	1352	124	2	735	93	2	735	77
2PC; $\{p_3, p_4\}$	336	2852	266	6	2600	231	3	1328	161	3	1328	134
2PC; $\{p_4, p_8\}$	576	1813	168	4	1237	112	2	575	75	2	648	66
ABP; $\{s_1, s_2\}$	64	628	27	8	574	21	4	289	18	4	304	12
ABP; $\{s_2, s_3\}$	64	1859	75	8	1832	70	4	946	55	4	943	37
ABP; $\{s_3, s_4\}$	32	374	18	4	353	13	2	188	12	2	192	8
ABP; $\{s_4, s_5\}$	32	3728	177	4	3638	170	2	1913	160	2	1833	93
ABP; $\{s_5, s_6\}$	64	449	27	8	412	21	4	199	18	4	201	11
ABP; $\{s_6, s_7\}$	64	1518	94	8	1481	87	4	769	80	4	752	47
2PC; $\{p_2, p_3, p_4\}$	2016	17478	1896	36	15646	1677	6	2693	719	6	2693	466
2PC; $\{p_3, p_4, p_8\}$	$^{79391}/_{-}$	101278	TO	36	23044	2498	6	4079	1064	6	3997	682
ABP; $\{s_1, s_2, s_3\}$	192	5641	226	24	5499	207	4	968	155	4	937	49
ABP; $\{s_2, s_3, s_4\}$	3072	23025	1470	48	19114	934	8	3639	722	8	3331	225
ABP; $\{s_3, s_4, s_5\}$	96	14651	748	12	15108	760	2	2599	567	2	2520	172
ABP; $\{s_4, s_5, s_6\}$	1536	14405	876	24	13269	686	4	2458	554	4	2215	151
ABP; $\{s_5, s_6, s_7\}$	192	4686	287	24	4559	268	4	809	241	4	748	57
2PC; $\{p_1, p_2, p_3, p_4\}$	$^{8064}/_{-}$	70250	TO	144	62280	11915	6	2770	2844	6	2719	1564
ABP; $\{s_1, s_2, s_3, s_4\}$	12288	90031	8143	192	76591	5458	8	3704	2931	8	3271	628
ABP; $\{s_3, s_4, s_5, s_6\}$	6144	59838	4777	96	52935	3543	4	2896	2655	4	2351	431
ABP; $\{s_4, s_5, s_6, s_7\}$	$^{1009}/_{-}$	108929	TO	$^{38}/_{96}$	111834	TO	$^{2}/_{4}$	10443	TO	4	8639	7480

Completion Enumeration. Table 1 presents the results for the completion enumeration use case and many-process synthesis scenario. Columns labeled *sol.* and *iter.* record the number of solutions (i.e., $|\mathsf{Sol}|$) and loop iterations of Algorithm 2 (i.e., the number of candidates $|\mathsf{Cand}|$, i.e., the number of times the SAT routine is called), respectively. Pilot experiments showed negligible variance across random seeds, so reported times are for one seed. TO denotes a timeout of 4 h, in which case $^{p}/_{q}$ means the tool produced p out of the total q solutions. For the dead opt. column, we know that $q = 24 \cdot n$, where n is the number of solutions/equivalence classes found by the permuter optimization and $24 = 4!$ is the number of isomorphisms for 4 states. Since the naive optimization produces equivalence classes, $q = n$ for the naive opt. column.

The results in Table 1 are consistent with our theoretical analyses. When there are 2 permutable states, the naive and permuter optimizations explore about half the number of candidates as the dead transitions method. For 3 permutable states, the optimized methods explore about $3! = 6$ times fewer candidates. For 4 permutable states, the optimized methods explore about $4! = 24$ times fewer candidates than the dead transitions method in the only experiment where the unoptimized method does not timeout. Notably, the permuter optimization does not timeout on any of these experiments.

Realizable 1-Completion. Table 2 presents the results for the realizable 1-completion use case (return the first solution found and stop) and many-process synthesis scenario. Our experiments and those of [4] suggest that there is more time variability for this task depending on the random seed provided to Z3. Thus, for Table 2 we run the tools for 10 different random seeds and report average times and number of iterations, rounded up. In one case (last row of Table 2), for a single seed out of the 10 seeds, the program timed out before finding a solution. As the true average is unknown in this case, we report it as TO.

Table 2. Many-Process Synthesis, Realizable 1-Completion

Case Study; A	unopt.		naive opt.		perm. opt.	
	iter.	time	iter.	time	iter.	time
2PC; $\{p_1, p_2\}$	199	19	157	20	157	17
2PC; $\{p_2, p_3\}$	483	47	429	55	426	46
2PC; $\{p_3, p_4\}$	798	72	696	84	666	69
2PC; $\{p_4, p_8\}$	380	37	319	44	311	34
ABP; $\{s_1, s_2\}$	111	4	110	7	100	4
ABP; $\{s_2, s_3\}$	220	9	205	13	200	9
ABP; $\{s_3, s_4\}$	106	5	102	7	105	5
ABP; $\{s_4, s_5\}$	1669	75	909	73	1202	60
ABP; $\{s_5, s_6\}$	102	5	95	8	102	5
ABP; $\{s_6, s_7\}$	507	28	294	28	294	17
2PC; $\{p_2, p_3, p_4\}$	440	48	590	147	561	89
2PC; $\{p_3, p_4, p_8\}$	954	94	861	205	796	121
ABP; $\{s_1, s_2, s_3\}$	332	12	225	36	240	13
ABP; $\{s_2, s_3, s_4\}$	2462	108	904	170	1028	64
ABP; $\{s_3, s_4, s_5\}$	2267	102	1040	219	819	52
ABP; $\{s_4, s_5, s_6\}$	2735	130	1513	333	1327	92
ABP; $\{s_5, s_6, s_7\}$	361	21	264	69	308	22
2PC; $\{p_1, p_2, p_3, p_4\}$	806	81	495	387	572	220
ABP; $\{s_1, s_2, s_3, s_4\}$	1957	85	1068	760	890	122
ABP; $\{s_3, s_4, s_5, s_6\}$	5425	261	1003	860	1601	234
ABP; $\{s_4, s_5, s_6, s_7\}$	16098	1088	TO	TO	4159	1158

Unrealizable 1-Completion. Table 3 presents the results for the unrealizable 1-completion use case and many-process synthesis scenario. For these experiments, we artificially modify the ABP *Sender* by completely removing state s_7, which results in no correct completion existing. A similar change is applied to *tx. man.*

in the case of 2PC. Thus, the tools explore the entire search space and terminate without finding a solution. As can be seen, the permuter optimization significantly prunes the search space and achieves considerable speedups.

Table 3. Many-Process Synthesis, Unrealizable 1-Completion

Case Study; A	unopt.		naive opt.		perm. opt.	
	iter.	time	iter.	time	iter.	time
2PC; $\{p_1, p_2\}$	3207	292	1658	206	1655	175
2PC; $\{p_2, p_3\}$	9792	978	4996	646	4982	552
2PC; $\{p_3, p_4\}$	14911	1527	7645	1053	7589	878
2PC; $\{p_4, p_8\}$	5123	494	2537	339	2555	282
ABP; $\{s_1, s_2\}$	1650	58	879	52	853	33
ABP; $\{s_2, s_3\}$	4300	173	2384	171	2374	106
ABP; $\{s_3, s_4\}$	327	13	173	11	164	7
ABP; $\{s_4, s_5\}$	3108	143	1592	130	1710	89
ABP; $\{s_5, s_6\}$	333	16	172	15	168	9
2PC; $\{p_2, p_3, p_4\}$	66088	TO	19717	10867	19850	9610
2PC; $\{p_3, p_4, p_8\}$	70586	TO	26343	TO	26516	14340
ABP; $\{s_1, s_2, s_3\}$	20858	1022	3705	798	3668	253
ABP; $\{s_2, s_3, s_4\}$	58974	4021	10516	2673	10496	1052
ABP; $\{s_3, s_4, s_5\}$	12323	596	2231	504	2167	146
ABP; $\{s_4, s_5, s_6\}$	11210	557	2104	491	1985	136
2PC; $\{p_1, p_2, p_3, p_4\}$	67659	TO	10365	TO	12308	TO
ABP; $\{s_1, s_2, s_3, s_4\}$	129264	TO	12096	TO	14739	TO
ABP; $\{s_3, s_4, s_5, s_6\}$	45056	2869	2466	2392	2004	339

6 Related Work

Synthesis of Distributed Protocols: Distributed system synthesis has been studied both in the reactive synthesis setting [23] and in the setting of discrete-event systems [29,30]. More recently, synthesis of distributed protocols has been studied using completion techniques in [2–4,17]. [2,4] study completion of finite-state protocols such as ABP but they do not focus on enumeration. [3] considers infinite-state protocols and focus on synthesis of symbolic expressions (guards and assignments). None of [2–4] propose any reduction techniques. We propose reduction modulo isomorphisms.

[17] studies synthesis for a class of parameterized distributed agreement-based protocols for which verification is efficiently decidable. Another version of the

paper [16] considers permutations of process indices. These are different from our permutations over process states.

Synthesis of parameterized distributed systems is also studied in [21] using the notion of *cutoffs*, which guarantee that if a property holds for all systems up to a certain size (the cutoff size) then it also holds for systems of any size. Cutoffs are different from our isomorphism reductions.

Bounded Synthesis: The bounded synthesis approach [12] limits the search space of synthesis by setting an upper bound on certain system parameters, and encodes the resulting problem into a satisfiability problem. Bounded synthesis is applicable to many application domains, including distributed system synthesis, and has been successfully used to synthesize systems such as distributed arbiters and dining philosophers [12]. Symmetries have also been exploited in bounded synthesis. Typically, such symmetries encode similarity of processes (e.g., all processes having the same state-transition structure, as in the case of dining philosophers). As such, these symmetries are similar to those exploited in parameterized systems, and different from our LTS isomorphisms.

Symmetry Reductions in Model-Checking: Symmetries have been exploited in model-checking [5]. The basic idea is to take a model M and construct a new model M_G which has a much smaller state space. This construction exploits the fact that many states in M might be functionally equivalent, in the sense of incoming and outgoing transitions. The key distinction between this work and ours is that our symmetries are over the space of models rather than the space of states of a fixed model. This distinction allows us to exploit symmetries for completion enumeration rather than model-checking.

Symmetry-Breaking Predicates: Symmetry-breaking predicates have been used to solve SAT [6], SMT [8], and even graph search problems [14], more efficiently. Our work is related in the sense that we are also trying to prune a search space. But our approach differs both in the notion of symmetry used (LTS isomorphism) as well as the application domain (distributed protocols). Moreover, rather than trying to eliminate all but one member of each equivalence class at the outset, say, by somehow adding a global (and often prohibitively large) symmetry-breaking formula Ξ to Φ, we do so *on-the-fly* for each candidate solution.

Canonical Forms: In program synthesis work [25], a candidate program is only checked for correctness if it is in some normal form. [25] is not about synthesis of distributed protocols, and as such the normal forms considered there are very different from our LTS isomorphisms. In particular, as with symmetry-breaking, the normal forms used in [25] are global, defined *a-priori* for the entire program domain, whereas our generalizations are computed on-the-fly. Moreover, the approach of [25] may still generate two equivalent programs as candidates (prior to verification), i.e., it does not satisfy our Property 2.

Sketching, CEGIS, OGIS, Sciduction: Completion algorithms such as GCG belong to the same family of techniques as sketching [27], counter-example guided inductive synthesis (CEGIS) [1,13,26,27], oracle-guided inductive synthesis (OGIS) [18], and sciduction [24].

7 Conclusions

We proposed a novel distributed protocol synthesis approach based on completion enumeration modulo isomorphisms. Our approach follows the *guess-check-generalize* synthesis paradigm, and relies on non-trivial optimizations of the generalization step that exploit state permutations. These optimizations allow to significantly prune the search space of candidate completions, achieving speedups of factors approximately 2 to 10 and in some cases completing experiments in minutes instead of hours. To our knowledge, ours is the only work on distributed protocol enumeration using reductions such as isomorphism.

As future work, we plan to employ this optimized enumeration approach for the synthesis of distributed protocols that achieve not only correctness, but also performance objectives. We also plan to address the question *where do the incomplete processes come from?* If not provided by the user, such incomplete processes might be automatically generated from example scenarios as in [2,4], or might simply be "empty skeletons" of states, without any transitions. We also plan to extend our approach to infinite-state protocols, as well as application domains beyond protocols, as Algorithm 2 is generic and thus applicable to a wide class of synthesis domains.

Acknowledgements. Derek Egolf's research has been initially supported by a Northeastern University PhD fellowship. This material is based upon work supported by the National Science Foundation Graduate Research Fellowship under Grant No. (1938052). 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. We thank Christos Stergiou for his work on the distributed protocol completion tool that we built upon. We also thank the anonymous reviewers for their helpful comments and feedback.

References

1. Alur, R., et al.: Syntax-guided synthesis. In: Formal Methods in Computer-Aided Design, FMCAD, pp. 1–17 (2013)
2. Alur, R., Martin, M., Raghothaman, M., Stergiou, C., Tripakis, S., Udupa, A.: Synthesizing finite-state protocols from scenarios and requirements. In: Yahav, E. (ed.) HVC 2014. LNCS, vol. 8855, pp. 75–91. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-13338-6_7
3. Alur, R., Raghothaman, M., Stergiou, C., Tripakis, S., Udupa, A.: Automatic completion of distributed protocols with symmetry. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9207, pp. 395–412. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21668-3_23

4. Alur, R., Tripakis, S.: Automatic synthesis of distributed protocols. SIGACT News **48**(1), 55–90 (2017)
5. Clarke, E.M., Emerson, E.A., Jha, S., Sistla, A.P.: Symmetry reductions in model checking. In: Hu, A.J., Vardi, M.Y. (eds.) CAV 1998. LNCS, vol. 1427, pp. 147–158. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0028741
6. Crawford, J.M., Ginsberg, M.L., Luks, E.M., Roy, A.: Symmetry-breaking predicates for search problems. In: Aiello, L.C., Doyle, J., Shapiro, S.C. (eds.) Proceedings of the Fifth International Conference on Principles of Knowledge Representation and Reasoning (KR 1996), Cambridge, Massachusetts, USA, 5–8 November 1996, pp. 148–159. Morgan Kaufmann (1996)
7. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
8. Dingliwal, S., Agarwal, R., Mittal, H., Singla, P.: CVC4-SymBreak: derived SMT solver at SMT competition 2019. CoRR, abs/1908.00860 (2019)
9. Egolf, D.: ATVA2023 artifact. https://github.com/egolf-cs/synge_reproducible
10. Egolf, D., Tripakis, S.: Decoupled fitness criteria for reactive systems. arXiv eprint arXiv:2212.12455 (2022)
11. Egolf, D., Tripakis, S.: Synthesis of distributed protocols by enumeration modulo isomorphisms. arXiv eprint arXiv:2306.02967 (2023)
12. 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
13. Gulwani, S., Polozov, O., Singh, R.: Program synthesis. Found. Trends Program. Lang. **4**(1–2), 1–119 (2017)
14. Heule, M.J.H.: The quest for perfect and compact symmetry breaking for graph problems. In: Davenport, J.H., et al. (eds.) 18th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2016, Timisoara, Romania, 24–27 September 2016, pp. 149–156. IEEE (2016)
15. Holzmann, G.: Design and Validation of Computer Protocols. Prentice Hall (1991)
16. Jaber, N., Jacobs, S., Kulkarni, M., Samanta, R.: Parameterized synthesis for distributed applications with consensus. https://www.cs.purdue.edu/homes/roopsha/papers/discoveri.pdf
17. Jaber, N., Wagner, C., Jacobs, S., Kulkarni, M., Samanta, R.: Synthesis of distributed agreement-based systems with efficiently-decidable parameterized verification. CoRR, abs/2208.12400 (2022)
18. Jha, S., Seshia, S.A.: A theory of formal synthesis via inductive learning. Acta Informatica **54**(7), 693–726 (2017). https://doi.org/10.1007/s00236-017-0294-5
19. Lamport, L.: Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley, Boston (2002)
20. Lynch, N.A.: Distributed Algorithms. Morgan Kaufmann Publishers Inc., San Francisco (1996)
21. 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
22. 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 (2015)
23. 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)

24. Seshia, S.A.: Sciduction: combining induction, deduction, and structure for verification and synthesis. In: Groeneveld, P., Sciuto, D., Hassoun, S. (eds.) The 49th Annual Design Automation Conference 2012, DAC 2012, San Francisco, CA, USA, 3–7 June 2012, pp. 356–365. ACM (2012)
25. Smith, C., Albarghouthi, A.: Program synthesis with equivalence reduction. In: Enea, C., Piskac, R. (eds.) VMCAI 2019. LNCS, vol. 11388, pp. 24–47. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-11245-5_2
26. Solar-Lezama, A., Tancau, L., Bodik, R., Seshia, S., Saraswat, V.: Combinatorial sketching for finite programs. SIGOPS Oper. Syst. Rev. **40**(5), 404–415 (2006)
27. Solar-Lezama, A.: Program sketching. Int. J. Softw. Tools Technol. Transf. **15**(5–6), 475–495 (2013). https://doi.org/10.1007/s10009-012-0249-7
28. Stergiou, C.: Distributed protocol completion tool. https://github.com/stavros7167/distributed_protocol_completion
29. Thistle, J.G.: Undecidability in decentralized supervision. Syst. Control Lett. **54**(5), 503–509 (2005)
30. Tripakis, S.: Undecidable problems of decentralized observation and control on regular languages. Inf. Process. Lett. **90**(1), 21–28 (2004)
31. Zave, P.: Reasoning about identifier spaces: how to make chord correct. IEEE Trans. Softw. Eng. **43**(12), 1144–1156 (2017)