# Recomposition: A New Technique for Efficient Compositional Verification

Ian Dardik
*Carnegie Mellon University*
Pittsburgh, PA, USA
idardik@andrew.cmu.edu

April Porter
*University of Maryland, College Park*
College Park, MD, USA
aporter3@terpmail.umd.edu

Eunsuk Kang
*Carnegie Mellon University*
Pittsburgh, PA, USA
eunsukk@andrew.cmu.edu

***Abstract*—Compositional verification algorithms are well-studied in the context of model checking. Properly selecting components for verification is important for efficiency, yet has received comparatively less attention. In this paper, we address this gap with a novel compositional verification framework that focuses on component selection as an explicit, first-class concept. The framework decomposes a system into components, which we then *recompose* into new components for efficient verification. At the heart of our technique is the *recomposition map* that determines how recomposition is performed; the component selection problem thus reduces to finding a good recomposition map. However, the space of possible recomposition maps can be large. We therefore propose heuristics to find a small portfolio of recomposition maps, which we then run in parallel. We implemented our techniques in a model checker for the TLA$^+$ language. In our experiments, we show that our tool achieves competitive performance with TLC–a well-known model checker for TLA$^+$–on a benchmark suite of distributed protocols.**

## I. Introduction

Model checking is an important tool for software, protocol, and algorithm development. *Compositional verification* is a paradigm in which a system is decomposed into components, which are then verified using a divide-and-conquer algorithm. To help model checking scale to large programs and specifications, compositional verification remains an important type of technique for combating the state explosion problem [1].

Most research papers on compositional verification assume that the components are pre-determined and focus solely on verification algorithms [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16]. However, *component selection*–that is, determining the set of decomposed components and the order in which they are verified–can greatly impact performance, in terms of both run time and state space size. Yet there are comparatively fewer model checking frameworks that investigate component selection, e.g. by automating decomposition [17], [18], [19]. Unfortunately, research into automated decomposition has seen limited success thus far; as Cobleigh et al. lament, decomposing a system is tough [17].

In this paper, we propose a new safety verification approach for symbolic specifications that is centered around component selection. In our approach, we begin by decomposing a system $S$ into components $C_1, \ldots, C_n$. Traditionally, a compositional verification algorithm is applied to these components to verify a system level property $P$, as shown in Fig. 1a. However, this verification problem may be less efficient than verifying the entire (monolithic) system directly without compositional techniques. Our key insight that addresses this shortcoming is to *recompose* the components into new components $D_1, \ldots, D_m$ that we verify instead. For example, Fig. 1b shows $D_1$ composed of $C_1$ and $C_3$ while $D_2$ is composed of $C_2$.

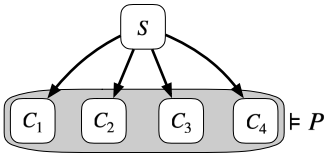
The choice of how to recompose is determined by a *recomposition map* that maps $C_i$'s to $D_j$'s. Recomposition maps make component selection an explicit, first-class concept and lie at the heart of our technique. We will show that, in practice, there often exists a recomposition map that results in a compositional verification problem that is more efficient than verifying the monolithic specification directly.

Additionally, we will show that our method is conducive to *specification reduction*. Specification reduction techniques, e.g. program slicing [20], [21], are generally considered separately from compositional verification. However, model checking with recomposition unites these two techniques under a single framework. For example, Fig. 1c shows a situation in which a partial recomposition map is used to reduce a specification with four-components to just the first three.

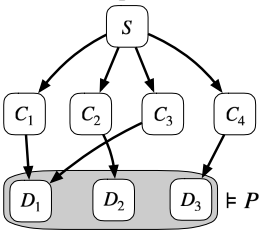
Ultimately, finding an efficient verification problem reduces to finding a suitable recomposition map. Therefore, we propose a technique for automatically inferring recomposition maps. We use heuristics to prune the large space of possible recomposition maps, which results in a small portfolio of maps that we run in parallel.

We have implemented our techniques in a model checker called "Recomp-Verify" for the TLA$^+$ language [22]. In order to bring compositional verification to TLA$^+$, we additionally propose a novel parallel composition operator for the language. We evaluate our techniques by comparing Recomp-Verify to TLC [23], a well-known model checker for TLA$^+$. We show that recomposition can lead to large savings in terms of verification time and the size of the explored state space.

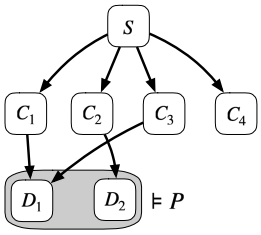
In summary, we make the following contributions: **(1) our main contribution, recomposition, which is a technique for efficient compositional verification,** (2) an automated method for finding efficient recomposition maps using parallelization and heuristics, (3) a definition for parallel composition for TLA$^+$ specifications, and (4) a prototype model checker Recomp-Verify that implements our algorithm, along with an evaluation of Recomp-Verify against TLC on a benchmark of




(a) Traditional compositional verification.




(b) Compositional verification with recomposition.




(c) Specification reduction in the recomposition method.

Fig. 1: Comparing traditional compositional verification against our recomposition method.

```
MODULE TwoPhase
1  CONSTANT RMs
2  VARIABLES msgs, rmState, tmState, tmPrepared
3  vars ≜ ⟨msgs, rmState, tmState, tmPrepared⟩

4  Init ≜
5     ∧ msgs = {}
6     ∧ rmState = [rm ∈ RMs ↦ "working"]
7     ∧ tmState = "init"
8     ∧ tmPrepared = {}

9  RcvPrepare(rm) ≜
10    ∧ [type ↦ "Prepared", theRM ↦ rm] ∈ msgs
11    ∧ tmState = "init"
12    ∧ tmPrepared' = tmPrepared ∪ {rm}
13    ∧ UNCHANGED ⟨msgs, tmState, rmState⟩

14 SndPrepare(rm) ≜
15    ∧ rmState[rm] = "working"
16    ∧ msgs' = msgs ∪ {[type ↦ "Prepared", theRM ↦ rm]}
17    ∧ rmState' = [rmState EXCEPT ![rm] = "prepared"]
18    ∧ UNCHANGED ⟨tmState, tmPrepared⟩

19 Next ≜
20    ∃ rm ∈ RMs :
21       ∨ SndPrepare(rm)
22       ∨ RcvPrepare(rm)
23          ⋮
24 Spec ≜ Init ∧ □[Next]_vars
```

Fig. 2: A monolithic encoding of the Two Phase Commit Protocol.

distributed protocols.

## II. Motivating Example

In this section, we describe the Two Phase Commit Protocol [24] to motivate our work and serve as a running example throughout the paper.

*1) Protocol Description:* In the Two Phase Commit Protocol, a *transaction manager* (TM) attempts to commit a transaction onto a pool of *resource managers* (RMs) in two phases. In the first phase, each RM starts in the *working* state as it attempts to commit the transaction. Any RM that can commit a transaction sends a *prepared* message to the TM. In the second phase, if every RM is prepared, the TM will issue a *commit* message to each RM; otherwise the TM will issue an *abort* message. The protocol assumes that the network can reorder, but not lose, messages. The key safety property is for each RM to remain *consistent*; i.e. no two RMs should disagree as to whether a transaction was committed or aborted.

*2) TLA⁺ Encoding:* In Fig. 2, we show only the first (prepare) phase of the $TwoPhase$ specification, which we will refer to as $TP$ for brevity. $TP$ is a *parameterized* protocol, meaning that the set of RMs in the protocol is given as input. In the $TP$ specification, the parameter is indicated on line 1 using the keyword CONSTANT.

$TP$ defines a symbolic transition system (STS) over four state variables, which are declared on line 2 in Fig. 2. The variable $rmState$ is the state of each RM, the variables $tmState$ and $tmPrepared$ hold the state of the TM, and $msgs$ is the set of messages each machine sends over the network. Line 24 formally declares the STS with initial predicate $Init$ and transition relation $Next$. We show two actions, $SndPrepare$ and $RcvPrepare$, on lines 14 and 9 respectively. In TLA⁺, actions are typically conjunctions of guards that specify when an action is enabled (lines 10-11 and 15) as well as primed variable expressions that specify transitions (lines 12 and 16-17). The UNCHANGED keyword on lines 13 and 18 indicate the frame conditions.

The key safety property for the Two Phase Commit Protocol is the invariant $Consistent$. We can encode this invariant as the following TLA⁺ formula:

$$\forall\, rm1,\, rm2 \in RMs :$$
$$\neg(rmState[rm1] = \text{"aborted"} \wedge\ rmState[rm2] = \text{"committed"})$$

*3) Model Checking TwoPhase:* The TLC model checker can prove that finite instances of $TP$ satisfy the property $Consistent$. A finite instance of a protocol substitutes a finite value for each parameter, e.g. a finite set of resource managers for $RMs$ in $TP$. TLC performs explicit state model checking, meaning that it enumerates every possible state in the transition system. For nine resource managers, TLC is able to prove $TP$ is safe after generating over 10 million states in nearly ten minutes. However, for ten resource managers, TLC fails to terminate in an hour after checking over 48 million states. In the following section, we will show how our approach can scale model checking $TP$ to ten resource managers.

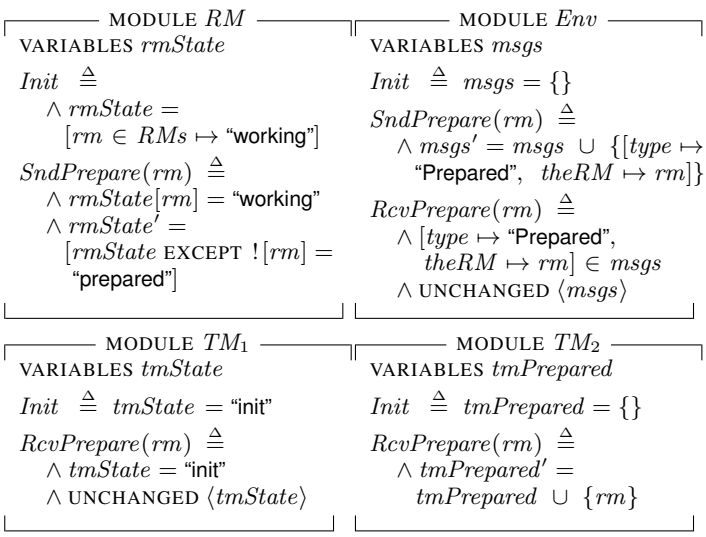



Fig. 3: A decomposition of $TP$. Standard operators such as $Spec$, $Next$, $vars$, etc. are omitted for brevity.

*4) Compositional Verification and Recomposition:* Consider the specifications $RM$, $Env$, $TM_1$, and $TM_2$ shown in Fig. 3. These specifications represent a decomposition of $TP$; that is, $TP$ is semantically equal to the parallel composition of the four specifications. We can generate a labeled transition system (LTS) for each of the four specifications in Fig. 3 and then use compositional verification techniques to answer the original model checking problem. For ten resource managers, this strategy enumerates a maximum of 261,002 states and terminates in 1 minute and 32 seconds.

The compositional verification problem above is more efficient than TLC, but *we can use recomposition to do even better*. Later, in example Ex. 2, we show how to use recomposition to identify new components that are *optimal* in terms of minimum run time for verification. In general, recomposition can provide large savings in terms of run time and state space. In Sec. VII, we show experimentally that recomposition can reduce a model checking problem by *millions* of states.

## III. Preliminaries

In this section we formally introduce *labeled transition systems (LTSs)*, the TLA$^+$ language, and the compositional verification technique that we consider in this paper. Throughout this paper, we will use calligraphic font when referring to LTS variables (e.g. $\mathcal{D}$) and normal font when referring to TLA$^+$ specifications (e.g. $S$).

### A. Labeled Transition Systems

A labeled transition system (LTS) $\mathcal{D}$ is a tuple $(Q, \alpha\mathcal{D}, \delta, I)$ where $Q$ is the set of states, $\alpha\mathcal{D}$ is the alphabet of $\mathcal{D}$, $\delta \subseteq Q \times \alpha\mathcal{D} \times Q$ is the transition relation, and $I$ is a set of initial states. $\alpha\mathcal{D}$ must be a subset of $\mathbb{A}$, where $\mathbb{A}$ is the universe of all possible actions across all possible LTSs. We let $Reach(\mathcal{D})$ be the set of reachable states in $\mathcal{D}$. We define the parallel composition ($\|$) over LTSs in the usual way by synchronizing on actions common to both alphabets and interleaving on all other actions.

$$
\begin{array}{ll}
spec ::= Spec \triangleq Init \wedge \Box[Next]_{vars} & conj ::= \wedge expr \mid \wedge expr\ conj \\
init ::= Init \triangleq conj & \\
next ::= Next \triangleq \exists x \in D : disj & disj ::= \vee expr \mid \vee expr\ disj \\
expr ::= \text{arbitrary TLA}^+\text{ expression} & op ::= id(p) \triangleq conj
\end{array}
$$

Fig. 4: Restricted TLA$^+$ grammar for this paper.

We define an action-based behavior $\sigma$ as an infinite sequence of actions, i.e. $\sigma \in \mathbb{A}^\omega$, and we let $\sigma_i$ denote the $i$th action in $\sigma$. We denote the action-based semantics of an LTS $\mathcal{D}$ as a set of action-based behaviors $[\![\mathcal{D}]\!]_\alpha \subseteq \mathbb{A}^\omega$. It is the case that $\sigma \in [\![\mathcal{D}]\!]_\alpha$ if and only if there exists a sequence of states $q_0, q_1, \ldots \in Q^\omega$ such that $q_0 \in I$ and, for each nonnegative index $i$, either (1) $\sigma_i \in \alpha\mathcal{D}$ and $(q_i, \sigma_i, q_{i+1}) \in \delta$, or (2) $\sigma_i \notin \alpha\mathcal{D}$ and $q_i = q_{i+1}$. Condition (2) allows for *stuttering*, a concept which we will introduce in Sec. III-B.

There are two methods for encoding a safety property as an LTS. The first method is creating an *error LTS* that includes an error state–which we refer to as the $\pi$ state–that acts as a sink for any action that causes a safety violation. The second method is creating a *property LTS* whose language defines the safe behaviors; property LTSs must be deterministic and must not include a $\pi$ state. Any error LTS can be converted to a property LTS using the construction described in [25]; specifically, steps two and three in section 3 for assumption generation. We define property satisfaction over property LTSs as follows: an LTS $\mathcal{D}$ satisfies a property LTS $\mathcal{P}$ ($\mathcal{D} \models \mathcal{P}$) exactly when $[\![\mathcal{D}]\!]_\alpha \subseteq [\![\mathcal{P}]\!]_\alpha$. Note that our LTS semantics (with stuttering) properly handles alphabet refinement, and therefore it is unnecessary to consider alphabet restriction [26] in our definition of property satisfaction.

### B. TLA$^+$

In this paper we will refer to a TLA$^+$ specification $S$ as a syntactic entity that consists of constants, variable and operator definitions, etc. in the format shown in Fig. 2.

The initial state predicate, transition relation, and specification declaration are named $Init$, $Next$, and $Spec$ respectively. In this paper, $Init$, $Next$, and $Spec$ are restricted to the syntax of $init$, $next$, and $spec$ given by the grammar in Fig. 4. In $next$, the domain $D$ does not contain state variables. We also restrict action definitions to the syntax of $op$, and no actions are referenced in the body of another action. In the grammar, $\Box$ is the *always* temporal operator. Expression $[Next]_{vars}$ is equal to $Next \vee (vars' = vars)$ and allows for *stuttering* states, i.e. consecutive states whose variables in $vars$ do not change.

We define several operators over a TLA$^+$ specification $S$. The scoping operator ! references definitions in $S$, e.g. $TP!SndPrepare$ refers to the $SndPrepare$ action of $TP$ in Fig. 2. The operators $\hat{\alpha}$ and $\alpha$ denote *symbolic actions*

and *concrete actions* respectively. Symbolic actions are the action names in a specification, while concrete actions are the actions that may occur in a finite instance. For example, let $TP_1$ be the finite instance of $TP$ with $RM = \{\text{“}rm1\text{”}\}$, then $\hat{\alpha}\,TP_1 = \hat{\alpha}\,TP = \{SndPrepare, RcvPrepare\}$ and $\alpha\,TP_1 = \{SndPrepare(\text{“}rm1\text{”}), RcvPrepare(\text{“}rm1\text{”})\}$. Additionally, we let $\beta S$ denote the set of state variables in a specification or an expression, e.g. $\beta TP = \{msgs, rmState, tmState, tmPrepared\}$. For an operator $* \in \{\hat{\alpha}, \alpha, \beta\}$ and a set of specifications $Z$, the notation $*Z$ is short-hand for the union of $*z$, for each specification $z \in Z$.

To define the semantics of a TLA$^+$ formula, we first define a state as an assignment to all state variables. Then, the semantics of a TLA$^+$ formula is a set of behaviors, where a behavior is an infinite sequence of states. We indicate state-based semantics of a TLA$^+$ formula $F$ as $[\![F]\!]_\beta$, the set of behaviors that satisfy $F$. For a TLA$^+$ specification $S$, we will often abbreviate $[\![S!Spec]\!]_\beta$ to simply $[\![S]\!]_\beta$. Given a TLA$^+$ property $P$, we say $S$ satisfies $P$ ($S \models P$) exactly when $[\![S]\!]_\beta \subseteq [\![P]\!]_\beta$.

We define the operator $\text{LTS}(S)$, which converts a TLA$^+$ specification $S$ into an LTS $\mathcal{S}$. $\text{LTS}(S)$ can be realized by generating the full state graph for $S$ and then labeling its edges with the concrete actions $\alpha S$ such that $\alpha\mathcal{S} = \alpha S$. Additionally, we define two operators for converting TLA$^+$ properties to an LTS. The first operator is $\text{LTS}(S, P)$, which builds an error LTS with $S$'s state space and actions, except violations of $P$ lead to a $\pi$ state. The second operator is $\text{PROP}(S, P)$, which builds a property LTS with $S$'s state space and actions that cannot cause a violation of $P$. $\text{PROP}(S, P)$ can be constructed from $\text{LTS}(S, P)$, as pointed out in Sec. III-A.

### C. CRA-Style Compositional Verification

In this paper, we consider *compositional reachability analysis* (CRA) [4], [8], [27] style compositional verification. Our recomposition framework requires a compositional verification algorithm that works for multiple components, and CRA-style techniques have reported success for verifying safety properties of multi-component systems [6].

CRA is used to check safety by composing the LTS for each component together in a hierarchical fashion; safety is proved if and only if the $\pi$ state is unreachable in the overall system. Such algorithms generally derive their divide-and-conquer efficiency from two optimizations: intermediate minimization and short-circuiting. The former involves minimizing the state space of the intermediate LTSs with respect to observational equivalence [28] during composition. The latter optimization, short-circuiting, occurs when a strict subset of components are needed for verification to succeed. In this case, the remaining components (outside the strict subset) can be skipped, and hence short-circuiting provides a *dynamic* form of specification reduction.

## IV. Parallel Composition in TLA$^+$

In this paper, we use two notions of parallel composition. The first is the usual parallel composition operator $\|$ that we define over LTSs in Sec. III-A. The second is a syntactic notion of parallel composition over TLA$^+$ specifications; the operator is syntactic because it is defined entirely based on TLA$^+$ syntax, and does not involve explicitly enumerating the state space. To avoid confusion, we will denote the TLA$^+$ parallel composition operator using $/\!/$. We will use the notation $/\!/Z$ to denote the composition over a set of specifications $Z$. We now define $/\!/$ in the usual way, by synchronizing common actions between specifications and interleaving all others actions.

**Definition 1** (Parallel Composition)**.** Let $S$ and $T$ be TLA$^+$ specifications with distinct state variables; we define $S /\!/ T$ as follows. First, $S /\!/ T$ contains exactly the constants and state variables in $S$ and $T$. Second, in $S /\!/ T$, we define $vars \triangleq (S!vars) \circ (T!vars)$, where $\circ$ is the sequence concatenation operator. Finally, in $S /\!/ T$, we define $Spec \triangleq Init \wedge [Next]_{vars}$ where $Init \triangleq S!\text{Init} \wedge T!\text{Init}$ and $Next$ is defined as follows:

$$\bigvee_{A \in \hat{\alpha}S \cup \hat{\alpha}T} \begin{cases} \exists d \in D : S!A(d) \wedge T!A(d) \\ \quad \text{if } A \in \hat{\alpha}S \text{ and } A \in \hat{\alpha}T \\ \exists d \in D : S!A(d) \wedge T!vars' = T!vars \\ \quad \text{if } A \in \hat{\alpha}S \text{ and } A \notin \hat{\alpha}T \\ \exists d \in D : T!A(d) \wedge S!vars' = S!vars \\ \quad \text{if } A \notin \hat{\alpha}S \text{ and } A \in \hat{\alpha}T \end{cases}$$

Notice that Def. 1 defines parallel composition in terms of TLA$^+$ syntax, and hence does not increase the expressivity of the language. While the operator itself is novel, this technique is briefly discussed by Lamport [22].

**Example 1.** By Def. 1, $TP = RM /\!/ Env /\!/ TM_1 /\!/ TM_2$. Furthermore, consider specifications $T_1$ and $T_2$ from Fig. 5. Notice that $TP = RM /\!/ T_1$, $T_1 = Env /\!/ T_2$, and $T_2 = TM_1 /\!/ TM_2$.

The following theorem shows that parallel composition behaves exactly as we expect if we convert a TLA$^+$ specification to an LTS. We prove this theorem using more general

MODULE $T_1$

VARIABLES
$msgs$, $tmState$, $tmPrepared$

$Init \triangleq$
$\wedge\ msgs = \{\}$
$\wedge\ tmState =$ “init”
$\wedge\ tmPrepared = \{\}$

$SndPrepare(rm) \triangleq$
$\wedge\ msgs' = msgs \cup$
$\{[type \mapsto$ “Prepared”,
$theRM \mapsto rm]\}$
$\wedge$ UNCHANGED
$\langle tmState, tmPrepared\rangle$

$RcvPrepare(rm) \triangleq$
$\wedge\ [type \mapsto$ “Prepared”,
$theRM \mapsto rm] \in msgs$
$\wedge\ tmState =$ “init”
$\wedge\ tmPrepared' =$
$tmPrepared \cup \{rm\}$
$\wedge$ UNCHANGED
$\langle msgs, tmState\rangle$

MODULE $T_2$

VARIABLES
$tmState$, $tmPrepared$

$Init \triangleq$
$\wedge\ tmState =$ “init”
$\wedge\ tmPrepared = \{\}$

$RcvPrepare(rm) \triangleq$
$\wedge\ tmState =$ “init”
$\wedge\ tmPrepared' =$
$tmPrepared \cup \{rm\}$
$\wedge$ UNCHANGED $\langle tmState\rangle$

Fig. 5: Intermediate decomposed specifications $T_1$ and $T_2$ in the $TP$ example.

semantics for TLA$^+$ specifications, namely action-state-based semantics. We include a proof in Appendix A.

**Theorem 1.** $[\![\text{LTS}(S /\!/ T)]\!]_\alpha = [\![\text{LTS}(S) \parallel \text{LTS}(T)]\!]_\alpha$.

## V. Model Checking with Recomposition

In this section, we propose our algorithm for verifying symbolic specifications. We begin by introducing the algorithm in Sec. V-A, followed by the details of decomposition in Sec. V-B and the details for compositional verification in Sec. V-C. Finally, we provide a correctness analysis of the algorithm in Sec. V-D.

### A. The Recomp-Verify Algorithm

*1) Algorithm Overview:* Our algorithm solves a model checking problem $S \models P$, where $S$ and $P$ are both written in TLA$^+$. The algorithm begins by decomposing $S$ into $n$ components $C_1, \ldots, C_n$, each of which is also a TLA$^+$ specification. The decomposition algorithm ensures two key properties upon termination: (P1) $S = C_1 /\!/ \cdots /\!/ C_n$ and (P2) the first component, $C_1$, contains all state variables that occur syntactically in $P$. Property (P1) ensures soundness of the decomposition, while property (P2) allows us to build the safety property $\mathcal{P}$ described in the following paragraph.

After decomposition, the algorithm *recomposes* the $C_i$ components into new components $D_P$ and $D_1, \ldots, D_m$. These new components define the following compositional verification problem that is equivalent to the original: $\text{LTS}(D_1) \parallel \ldots \parallel \text{LTS}(D_m) \models \mathcal{P}$, where $\mathcal{P} = \text{PROP}(D_P, P)$. For $\text{PROP}(D_P, P)$ to be well-formed, $D_P$ must contain every state variable that occurs in $P$. Therefore, we require $D_P$ to be composed of $C_1$, as $C_1$ must contain every state variable that occurs in $P$ by property (P2) of decomposition. We formally capture this requirement, as well as the choice of how to perform recomposition, in the following definition.

**Definition 2** (Recomposition Map)**.** A *recomposition map* is a surjective function $f : \{C_1, \ldots, C_n\} \to \{d_P, d_1, \ldots, d_m\}$ such that $f(C_1) = d_P$.

In Def. 2, the $d_j$'s in the co-domain are intended as a placeholder for constructing each $D_j$. In particular, we will define each recomposed component as $D_j = /\!/ f^{-1}(d_j)$, the parallel composition of one or more $C_i$ components. Therefore, the restriction $f(C_1) = d_P$ implies that $D_P$ will be composed of $C_1$ as intended. Finally, once each $D_j$ is constructed, we solve the compositional verification problem.

*2) Algorithm Details:* We present our model checking algorithm in Alg. 1. The algorithm accepts several inputs, including a *recomposition strategy*. A recomposition strategy $\rho$ is a function that maps $C_i$ components to a pair $(f, m)$, where $f$ is a recomposition map and $m$ is the number of $D_j$ components. In other words, the recomposition strategy determines which recomposition map is used. In this section we assume $\rho$ is given; we discuss recomposition strategy selection in Sec. VI.

Alg. 1 begins by decomposing $S$ into components on line 1; we provide more detail for decomposition in Sec. V-B. Next, on lines 3-5, we perform recomposition using the recomposition map $f$. On line 3, we define $D_P$ to be the parallel composition of each $C_i$ component in the pre-image $f^{-1}(d_P)$. Similarly, on line 5, we define each $D_j$ to be the parallel composition of each $C_i$ component in the pre-image $f^{-1}(d_j)$. Finally, on line 6, we solve the compositional verification problem for the recomposed components ($D_j$'s); we provide more detail for this step in Sec. V-C.

**Algorithm 1** RECOMP-VERIFY

**Input:** Specification $S$, property $P$, recomposition strategy $\rho$
**Output:** If $S \models P$

1: $C_1, \ldots, C_n = \text{DECOMPOSE}(S, P)$
2: $f, m \leftarrow \rho(C_1, \ldots, C_n)$
3: $D_P \leftarrow /\!/ f^{-1}(d_P)$ ▷ $f^{-1}(d_P) \subseteq \{C_1, \ldots, C_n\}$
4: **for** $j \in \{1, \ldots, m\}$ **do**
5: $\quad D_j \leftarrow /\!/ f^{-1}(d_j)$ ▷ $f^{-1}(d_j) \subseteq \{C_2, \ldots, C_n\}$
6: **return** $\text{COMP-VERIFY}(D_1, \ldots, D_m, D_P, P)$

**Example 2.** In this example we analyze Alg. 1 given the input $TP$, $Consistent$, and a hand-crafted optimal recomposition strategy $\rho_{opt}$. Line 1 of Alg. 1 will produce the components $RM$, $Env$, $TM_1$, and $TM_2$ from Fig. 3. $\rho_{opt}$ chooses $m = 2$ and $f$ such that $f(RM) = d_P$, $f(Env) = f(TM_1) = d_1$, and $f(TM_2) = d_2$. Recomposition (lines 3-5) reduces the original model checking problem to $\text{LTS}(Env /\!/ TM_1) \parallel \text{LTS}(TM_2) \models \text{PROP}(RM, Consistent)$, which we solve on line 6. Whereas the example in Sec. II verifies four specifications (for $RM$, $Env$, $TM_1$, $TM_2$), this example verifies three (for $RM$, $Env /\!/ TM_1$, $TM_2$). The strategy $\rho_{opt}$ in this example reduces the maximum state space by 1,027 states and also improves the model checking time from 1 minute 32 seconds to 51 seconds.

*3) Static Specification Reduction:* Total recomposition maps apply verification to every component, hence considering the entire specification $S$. However, we are able to statically compute a subset of the components that are necessary for verification. We therefore allow recomposition maps to be partial, defined only for the components that are necessary for verification. Partial recomposition maps are a *static specification reduction* technique, akin to program slicing [20], [21].

The subset of necessary components is those whose alphabets may affect–either directly or indirectly–the actions of $C_1$, and therefore may prevent the entire system from reaching an error. More formally, the subset of components is $\bigcup_i X_i$, where $X_0 = \{C_1\}$ and $X_{i+1} = \{C_j \mid \hat{\alpha} C_j \cap \hat{\alpha} X_i \neq \emptyset\}$. In Appendix B, we show that it is only necessary to consider the first $n + 1$ terms when computing the union of the sequence. Intuitively, $X_1$ is the set of components that may directly prevent $C_1$ from reaching an error, while $X_2$, $X_3$, etc. may indirectly prevent an error.

**Example 3.** We now introduce $TPCounter$, an extension to $TP$. $TPCounter$ is identical to $TP$, except it includes one more state variable $counter$ and one more action $Increment$. In the initial state, $counter$ is equal to zero. Each action

that $TPCounter$ inherits from $TP$ leaves $counter$ unchanged, while $Increment$ increments $counter$ by one and leaves all other state variables unchanged. The $Increment$ action is always enabled, and therefore $TPCounter$ is an infinite-state protocol.

Consider model checking $TPCounter \models Consistent$ with Alg. 1. Decomposition (line 1) produces five components: $RM$, $Env$, $TM_1$, $TM_2$, and $Counter$, where $Counter$ has one state variable $counter$ and one action $Increment$. $Counter$ is the only specification with the action $Increment$ and, therefore, does not synchronize with the actions in the other four specifications. Therefore, $Counter$ cannot affect the safety of $C_1$. Formally, $X_0 = \{RM\}$, $X_1 = \{RM, Env\}$, $X_2 = \{RM, Env, TM_1, TM_2\}$, $X_3 = X_2$, etc. so $Counter$ is not a necessary component. $Counter$ may therefore be safely omitted from the domain of any recomposition map, which will cause Alg. 1 to successfully terminate.

Thm. 2 formally shows that it is sound and complete to reduce verification to the necessary components; we prove this theorem in Appendix B.

**Theorem 2.** *$S \models P$ if and only if $/\!/(\bigcup_i X_i) \models P$.*

### B. Decomposition

In this section, we present an algorithm for decomposing a symbolic specification $S$ into $n$ components $C_1 \dots C_n$. Our algorithm guarantees the following two properties: (P1) $S = C_1 /\!/ \cdots /\!/ C_n$ and (P2) $\beta P \subseteq \beta C_1$. We provide a correctness argument for these two properties in Sec. V-D.

*1) Decomposition Algorithm:* Each step of the algorithm splits a specification $T_i$ into two specifications $C_{i+1}$ and $T_{i+1}$ such that $T_i = C_{i+1} /\!/ T_{i+1}$. We note the following two corner cases: $T_0 = S$ and $C_n = T_{n-1}$. The algorithm splits a specification across two phases: *state variable partitioning* and *specification slicing*. The former partitions the state variables of $T_i$ into two sets $V_C$ and $V_T$, while the latter slices $T_i$ into $C_{i+1}$ and $T_{i+1}$ that contain the variables $V_C$ and $V_T$ respectively. We present the algorithm in Alg. 2. We now explain state variable partitioning and specification slicing in detail across the following two sections.

**Example 4.** We explain Alg. 2 given $TP$ and $Consistent$. The algorithm begins with the partition $V_C = \{rmState\}$ and $V_T = \{msgs, rmState, rmPrepared\}$ on line 1; we explain partitioning in Sec. V-B2. Next, on lines 6-7, the algorithm slices $TP$ into $RM$ (Fig. 3) and a new specification $T_1$ (see Appendix C, Fig. 5). The state variables of $T_1$ are subsequently partitioned into $V_C = \{msgs\}$ and $V_T = \{rmState, rmPrepared\}$ on line 9. The algorithm continues in this fashion until $V_T = \emptyset$, i.e. no partition is possible. The algorithm will then exit the loop and return the components $RM$, $Env$, $TM_1$, $TM_2$ on line 12.

*2) State Variable Partitioning:* Given a specification $T_i$, the partitioning phase partitions the variables $\beta T_i$ into two sets $V_C$ and $V_T$. The partition procedure appears twice in Alg. 2. The first occurrence, on line 1, determines the state variables that will appear in $C_1$; therefore, to uphold property (P2), we partition on $\beta P$. In the second appearance, on line 9, we choose just *one* variable in attempt to produce as many components as possible (ideally, one component per state variable). We are free to choose the one variable nondeterministically because the order of decomposed components is inconsequential; this is due to the fact that the ordering is ultimately determined by a recomposition map in Alg. 1.

The partition procedure in Alg. 2 also guarantees that the state variables in each partition will constitute a well-formed slice according to the grammar in Fig. 4. For example, if a specification contains the expression $a = b + 1$, then $a$ and $b$ should be grouped together into the same partition. To accomplish this, we let $V_C$ be $V$ *plus* any variables that occur within the same expression, repeated until fix-point. More formally, we let $V_C = \textsc{Fix}(\textsc{Occurs}_S, V)$ (line 14), where FIX invokes the $\textsc{Occurs}_S$ procedure, initially on $V$, until a fix-point is reached. Finally, we choose $V_T$ to be the remainder of the state variables in $T_i$ (line 15).

**Example 5.** Notice that $\beta Consistent = \{rmState\}$ and $\textsc{Fix}(\textsc{Occurs}_{TP}, \{rmState\}) = \{rmState\}$. Therefore, the first partition (line 1) will be $V_C = \{rmState\}$ and $V_T = \{msgs, tmState, tmPrepared\}$. In the second partition (lines 8-9), we arbitrarily choose $v = msgs$, which results in $V_C = \{msgs\}$ and $V_T = \{tmState, tmPrepared\}$.

**Algorithm 2** DECOMPOSE

**Input:** Specification $S$, Safety Property $P$
**Output:** $C_1, \dots, C_n$ with properties (P1) and (P2)

1: $V_C, V_T \leftarrow \textsc{Partition}(S, \beta P)$
2: **if** $V_T = \emptyset$ **then**
3: **return** $S$
4: $T_0 \leftarrow S$, $i \leftarrow 0$
5: **while** $V_T \neq \emptyset$ **do**
6: $C_{i+1} \leftarrow \textsc{Slice}(T_i, V_C)$
7: $T_{i+1} \leftarrow \textsc{Slice}(T_i, V_T)$
8: $v \in \beta T_{i+1}$ ▷ Nondeterministically choose a variable
9: $V_C, V_T \leftarrow \textsc{Partition}(T_{i+1}, \{v\})$
10: $i \leftarrow i + 1$
11: $n \leftarrow i + 1$, $C_n \leftarrow T_i$
12: **return** $C_1, \dots, C_n$
13: **procedure** $\textsc{Partition}(T, V)$
14: $V_C \leftarrow \textsc{Fix}(\textsc{Occurs}_T, V)$
15: $V_T \leftarrow \beta S - V_C$
16: **return** $V_C, V_T$
17: **procedure** $\textsc{Occurs}_S(V)$
18: **return** $\bigcup_{A \in \hat{\alpha} S} \{\beta c \mid c \in \text{Conj}(A) \text{ and } \beta c \cap V \neq \emptyset\}$
19: **procedure** $\textsc{Fix}(op, X)$
20: $Y \leftarrow X \cup op(X)$
21: **if** X = Y **then return** $X$
22: **return** $Y \cup \textsc{Fix}(op, Y)$

*3) Specification Slicing:* The specification slicing phase restricts a specification $T_i$ to a given subset of its variables $V$. Slicing can be seen as the inverse of parallel composition. For example, consider a system specification $M$ with action $Action$ and state variables $var_1$ and $var_2$:

$$Action \triangleq \begin{array}{l} \wedge\, var_1{}' = \text{“val1”} \\ \wedge\, var_2{}' = \text{“val2”} \end{array}$$

Given the variable partition $\{var_1\}$, $\{var_2\}$, we can view $M$ as the composition of two components $M_1$ and $M_2$ that respectively define: $Action \triangleq var_1{}' = \text{“val1”}$ and $Action \triangleq var_2{}' = \text{“val2”}$. In particular, we have $M_1 = \text{SLICE}(M, \{var_1\})$, $M_2 = \text{SLICE}(M, \{var_2\})$, and $M = M_1 /\!/ M_2$. In the $TP$ example, this corresponds to $TP = RM /\!/ T_1$ in Ex. 4. We include more details on slicing, including the definition for the slicing procedure, in Appendix C.

### C. Compositional Verification

We present a CRA-style compositional verification algorithm in Alg. 3. The algorithm works by iteratively composing the LTS for each component $D_j$ together (line 5) until the $\pi$ state becomes unreachable, in which case verification succeeds (lines 3 and 7). If the $\pi$ state remains reachable by the end of the algorithm, however, then we report a failure (line 8). The algorithm performs intermediate minimization on lines 1 and 5. In general, there are many options for which components–or composition of components–to minimize [12]. We choose to only minimize components because we observed that minimizing the composition of components was generally slow. In essence, this algorithm is an abstraction-refinement loop where each new component lowers the abstraction by introducing more state variables.

**Example 6.** Consider $TP$ with ten resource managers and the optimal mapping $f$ from Ex. 2, where $D_P = RM$, $D_1 = Env /\!/ TM_1$, and $D_2 = TM_2$. Line 1 of Alg. 3 will generate an LTS for $D_P$ with 477,454 states, including a $\pi$ state. Minimization reduces $D_P$ to 13,291 states. Due to a reachable $\pi$ state, the algorithm proceeds into the loop on line 4. Next, on line 5, the algorithm generates an LTS for $D_1$ with 3,072 states, which reduces to 1,026 states after minimization. Composing this LTS with $\mathcal{D}$ (line 5) retains the $\pi$ state (line 6) so we loop again. The algorithm continues in this fashion until a $\pi$ state is no longer reachable, and we return a positive answer (line 6 and 7). A maximum of 481,550 states are needed in memory at once.

### D. Correctness Analysis

In this section we show that Alg. 1 is sound but not complete. Proving the soundness of Alg. 1 relies on the correctness of decomposition (Sec. V-B) and compositional verification (Sec. V-C). Therefore, we begin by presenting two lemmas; Lem. 1 for decomposition correctness and Lem. 2 for compositional verification correctness.

**Lemma 1.** *Algorithm 2 ensures (P1) $S = C_1 /\!/ \cdots /\!/ C_n$ and (P2) $\beta C_1 \subseteq \beta P$ upon termination.*

**Algorithm 3** COMP-VERIFY

**Input:** $D_1, \ldots, D_m, D_P, P$
**Output:** If $\text{LTS}(D_1) \parallel \ldots \parallel \text{LTS}(D_m) \models \mathcal{P}$

1: $\mathcal{D} \leftarrow Min(\text{LTS}(D_P, P))$ ▷ $\mathcal{P} = \text{LTS}(D_P, P)$
2: **if** $\pi \notin Reach(\mathcal{D})$ **then**
3: **return** *true*
4: **for** $j \in \{1, \ldots, m\}$ **do**
5: $\mathcal{D} \leftarrow \mathcal{D} \parallel Min(\text{LTS}(D_j))$
6: **if** $\pi \notin Reach(\mathcal{D})$ **then**
7: **return** *true*
8: **return** *false*

*Proof.* Sketch. We prove property (P1) by establishing the following loop invariant in Alg. 2 on line 5: $S = C_1 /\!/ \cdots /\!/ C_i /\!/ T_i$. The proof for property (P2) follows in two steps. First, $\beta P \subseteq V_C$ because $V_C$ (in the first partition) is defined by a monotonically increasing operation (FIX) on $\beta P$. Second, $\beta P \subseteq \beta C_1$ because $C_1 = \text{SLICE}(S, V_C)$ will contain exactly the state variables in $V_C$. □

**Lemma 2.** $\text{LTS}(D_1) \parallel \ldots \parallel \text{LTS}(D_m) \models \mathcal{P}$ *if and only if there exists a* $k \in \{0 \ldots m\}$ *such that* $\pi \notin Reach(\text{LTS}(D_P, P) \parallel \text{LTS}(D_1) \parallel \ldots \parallel \text{LTS}(D_k))$.

*Proof.* Sketch. The forwards case ($\Rightarrow$) follows by choosing $k = m$. For the backwards case ($\Leftarrow$), we assume that $k$ is given such that $0 \leq k \leq m$ and $\pi \notin Reach(\text{LTS}(D_P, P) \parallel \text{LTS}(D_1) \parallel \ldots \parallel \text{LTS}(D_k))$. Notice that none of the LTSs $\text{LTS}(D_{k+1}), \ldots, \text{LTS}(D_m)$ contain a reachable $\pi$ state, and therefore neither will $Reach(\text{LTS}(D_P, P) \parallel \text{LTS}(D_1) \parallel \ldots \parallel \text{LTS}(D_m))$. □

Finally, Thm. 3–coupled with Lem. 2–shows that Alg. 1 is sound.

**Theorem 3.** $S \models P \iff \text{LTS}(D_1) \parallel \ldots \parallel \text{LTS}(D_m) \models \mathcal{P}$.

*Proof.*

$$\begin{aligned}
& S \models P && (1)\\
\iff & \pi \in Reach(\text{LTS}(S, P)) && (2)\\
\iff & \pi \in Reach(\text{LTS}(C_1 /\!/ \cdots /\!/ C_n, P)) && (3)\\
\iff & \pi \in Reach(\text{LTS}(D_P /\!/ D_1 /\!/ \cdots /\!/ D_m, P)) && (4)\\
\iff & \pi \in Reach(\text{LTS}(D_P, P) \parallel \ldots \parallel \text{LTS}(D_m, P)) && (5)\\
\iff & \pi \in Reach(\text{LTS}(D_P, P) \parallel \ldots \parallel \text{LTS}(D_m)) && (6)\\
\iff & \text{LTS}(D_1) \parallel \ldots \parallel \text{LTS}(D_m) \models \text{PROP}(D_P, P) && (7)
\end{aligned}$$

Biconditional (2) holds because $P$ is a safety property, (3) by Lem. 1 property (P1), (4) by the definition for each $D_j$ (and Thm. 2 in the case that $f$ is partial), (5) by Thm. 1, (6) by Lem. 1 property (P2) and Def. 2 ($f(C_1) = d_P$), and (7) because $P$ is a safety property. Finally, the theorem follows because $\mathcal{P} = \text{PROP}(D_P, P)$. □

While the above results show that Alg. 1 is sound, the algorithm is not complete, even if we limit $S$ to be a finite-state specification. This is because a given component $D_j$ may be

infinite-state, in which case $\text{LTS}(D_j)$ will fail to terminate on line 1 or 5 in Alg. 3. We address this limitation in Sec. VI-B by using a portfolio of strategies that includes the *monolithic strategy*.

## VI. Choosing Efficient Recomposition Maps

In this section, we address the problem of designing recomposition strategies, i.e. choosing efficient recomposition maps. Rather than finding a single recomposition strategy, we propose running Alg. 1 with a portfolio of strategies in parallel. The primary challenge is determining which strategies to use, since the number of possible recomposition maps grows large as the number of components increases. We therefore propose a heuristic for pruning the search space of recomposition maps in Sec. VI-A. We then choose a small portfolio of strategies based on this heuristic in Sec. VI-B.

### A. Recomposition Map Reduction Heuristic

Any heuristic for pruning the search space of recomposition maps should be tailored to the compositional verification algorithm being used. Since we use a CRA style verification algorithm, we design our heuristic to find component orderings that can take advantage of short-circuiting. In particular, the heuristic identifies recomposition maps that order $D_j$ components that are least likely to be necessary for verification *last*.

Our heuristic is to choose recomposition maps that respect the *data flow* partial order $\preccurlyeq$ over the $C_i$ components. This is a novel partial order that attempts to find dynamic specification reduction–i.e. short-circuiting–by refining our static specification reduction scheme. Intuitively, the partial order will order the components based on how far removed their state variables are from impacting verification.

More formally, we compute the data flow partial order based on the sequence $\{X_i\}$ introduced in Sec. V-A3. This sequence cumulatively captures the components that may interact–either directly or indirectly–with the actions of $C_1$. First, we build a new sequence $\{E_i\}$ defined as $E_0 = X_0$ and $E_{i+1} = X_{i+1} \setminus X_i$. While the $X_i$'s are cumulative, each $E_i$ captures only the additional components in each $X_i$. Intuitively, the components in $E_i$ are $i$ steps removed from affecting the variables in $C_1$, and hence $i$ steps removed from impacting verification (by property (P2) of decomposition).

Second, we build a sequence $\{F_i\}$ that captures the *data flow* from each component in $E_i$ to the components in $E_{i+1}$. We define $F_0 = \emptyset$ and:

$$F_{i+1} = \left\{ (C_j, C_k) \;\middle|\; \begin{array}{c} C_j \in E_i \text{ and } C_k \in E_{i+1} \\ \text{and } \hat{\alpha} C_j \cap \hat{\alpha} C_k \neq \emptyset \end{array} \right\}$$

Finally, let $F = \bigcup_i F_i$; we define the data flow partial order $\preccurlyeq$ to be the reflexive transitive closure of $F$. In Appendix D, we show formally that the partial order refines the static specification reduction scheme from Sec. V-A3.

**Example 7.** For $TP$ and $Consistent$, $E_0 = \{RM\}$, $E_1 = \{Env\}$, and $E_2 = \{TM_1, TM_2\}$. Moreover, $F = \{(RM, Env), (Env, TM_1), (Env, TM_2)\}$ and the data flow partial order is the reflexive transitive closure of this set. Intuitively, the partial order shows that $TM_1$ and $TM_2$ can only affect the variables in $RM$–i.e. the variables $\beta Consistent$ needed for verification–*indirectly* by interacting with the $Env$ component.

To further reduce the search space of maps, we extend the data flow partial order to a total order $\leqslant$ , i.e. $\preccurlyeq \subseteq \leqslant$ . We build the total order by breaking ties between incomparable components $C_i$ and $C_j$ by requiring $C_i \leqslant C_j$ if and only if $C_i$'s state variables have fewer syntactic appearances than $C_j$'s in the original specification $S$.

### B. Choosing a Portfolio of Strategies

In this section, we describe four recomposition strategies that comprise our portfolio. For simplicity, we describe the strategies assuming that the components $C_1, \ldots, C_n$ have been reordered according to the total order $\leqslant$ described in Sec. VI-A. The four strategies are (S1) the identity strategy, in which $m = n - 1$ and $f(C_i) = d_i$ for all $i$, (S2) a "bottom heavy" strategy in which we choose $m = 1$ and $f$ such that $f(C_1) = d_P$ and $f(C_i) = d_1$ for all $i > 1$, (S3) a "top heavy" strategy in which we choose $m = 1$ and $f$ such that $f(C_n) = d_1$ and $f(C_i) = d_P$ for all $i < n$, and (S4) the *monolithic* strategy, where $m = 0$ and $f(C_i) = d_P$ for all $i$.

**Example 8.** The state variables in $TM_2$ occur fewer times syntactically than the variables of $TM_1$ within $TP$, and hence the total ordering resulting from Sec. VI-A is $RM, Env, TM_2, TM_1$. Then:

1) In the identity strategy, $m = 3$, $f(RM) = d_P$, $f(Env) = d_1$, $f(TM_2) = d_2$, and $f(TM_1) = d_3$.
2) In the bottom heavy strategy, $m = 1$, $f(RM) = d_P$, and $f(Env) = f(TM_2) = f(TM_1) = d_1$.
3) In the top heavy strategy, $m = 1$, $f(RM) = f(Env) = f(TM_2) = d_P$, and $f(TM_1) = d_1$.
4) In the monolithic strategy, $m = 0$ and $f(RM) = f(Env) = f(TM_2) = f(TM_1) = d_P$.

As a note regarding the correctness analysis from Sec. V-D, including the monolithic strategy in the portfolio ensures termination. Therefore, our parallel approach is complete for finite state specifications $S$.

## VII. Experimental Results

### A. Implementation

We have created a model checker called Recomp-Verify that can verify safety for TLA$^+$ specifications. The model checker is a prototype research tool that implements Alg. 1 in the Python, Java, and Kotlin programming languages. The model checker also supports running multiple instances of Alg. 1 in parallel, and returns the first result to finish. Our tool is available in a public repository [29].

### B. Experiments

We evaluate Recomp-Verify against TLC on a benchmark of distributed protocols [30], plus the tla-twophase-counter protocol that we introduce in Ex. 3. Our evaluation is driven

| | | | | Recomp-Verify$^1$ | | TLC$^1$ | | Recomp-Verify$^4$ | | | TLC$^4$ | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Name | n | m | k | States | Time | States | Time | States | Time | Strat. | States | Time |
| tla-consensus-3 | 1 | 0 | 0 | 4 | 1s | 4 | 1s | 4 | 1s | S4 | 4 | 1s |
| tla-tcommit-3 | 1 | 0 | 0 | 34 | 1s | 34 | 1s | 34 | 1s | S4 | 34 | 1s |
| i4-lock-server-2-2 | 1 | 0 | 0 | 9 | 1s | 9 | 1s | 9 | 1s | S4 | 9 | 1s |
| ex-quorum-leader-election-6 | 2 | 1 | 1 | 117,671 | 39s | 121,111 | **4s** | 121,111 | 5s | S4 | 121,111 | **2s** |
| pyv-toy-consensus-forall-6-6 | 3 | 1 | 1 | 117,671 | 33s | 121,111 | **4s** | 121,111 | 5s | S4 | 121,111 | **2s** |
| tla-simple-5 | 1 | 0 | 0 | 723 | 1s | 723 | 1s | 723 | 1s | S4 | 723 | 1s |
| ex-lockserv-automaton-20 | 5 | 1 | 0 | 61 | **2s** | - | TO | 61 | **3s** | S3 | - | TO |
| tla-simpleregular-5 | 1 | 0 | 0 | 2,524 | 2s | 2,524 | **1s** | 2,524 | 1s | S4 | 2,524 | 1s |
| pyv-sharded-kv-3-3-3 | 3 | 0 | 0 | 10,648 | 5s | 10,648 | **2s** | 10,648 | 2s | S4 | 10,648 | **1s** |
| pyv-lockserv-20 | 5 | 1 | 0 | 61 | **1s** | - | TO | 61 | **2s** | S3 | - | TO |
| tla-twophase-9 | 4 | 2 | 2 | 145,176 | **19s** | 10,340,352 | 9m41s | 145,691 | **31s** | S1 | 10,340,352 | 2m36s |
| tla-twophase-10 | 4 | 2 | 2 | 481,550 | **1m8s** | - | TO | 482,577 | **1m36s** | S1 | - | TO |
| tla-twophase-counter-9 | 5 | 2 | 2 | 145,176 | **19s** | - | TO | 145,691 | **31s** | S1 | - | TO |
| i4-learning-switch-4-3 | 1 | 0 | - | - | TO | 1,344,192 | **5m55s** | 1,344,192 | 5m55s | S4 | 1,344,192 | **1m37s** |
| ex-simple-decentralized-lock-4 | 2 | 0 | 0 | 20 | 2s | 20 | **1s** | 20 | 1s | S4 | 20 | 1s |
| i4-two-phase-commit-7 | 4 | 2 | 2 | 151,348 | **26s** | 10,016,384 | 3m38s | 184,112 | **27s** | S3 | 10,016,384 | 53s |
| pyv-consensus-wo-decide-4 | 5 | 2 | 1 | 32,816 | **9s** | - | TO | 32,953 | **10s** | S3 | - | TO |
| pyv-consensus-forall-4-4 | 6 | 1 | 0 | 33,545 | **8s** | - | TO | 33,545 | **9s** | S3 | - | TO |
| pyv-learning-switch-trans-3 | 2 | 1 | 0 | 729 | **5s** | - | TO | 729 | **6s** | S1 | - | TO |
| pyv-learning-switch-sym-2 | 2 | 1 | 0 | 4 | 2s | 1,344 | **1s** | 1,344 | 1s | S4 | 1,344 | 1s |
| pyv-sharded-kv-no-lost-keys-3-3-3 | 2 | 0 | 0 | 9,261 | 4s | 27 | **1s** | 9,261 | 2s | S4 | 9,261 | **1s** |
| ex-naive-consensus-4-4 | 3 | 1 | 1 | 824 | 2s | 1,001 | **1s** | 1,001 | 2s | S4 | 1,001 | **1s** |
| pyv-client-server-ae-4-2-2 | 2 | 1 | 1 | 352,145 | **42s** | 2,039,392 | 1m36s | 352,145 | 49s | S1 | 2,039,392 | **28s** |
| pyv-client-server-ae-2-4-2 | 2 | 1 | 1 | 894,437 | 2m18s | 2,387,032 | **1m16s** | 2,387,032 | 1m26s | S4 | 2,387,032 | **22s** |
| ex-simple-election-6-7 | 3 | 1 | 0 | 267,590 | **1m20s** | 2,900,256 | 3m7s | 267,590 | 1m22s | S3 | 2,900,256 | **54s** |
| pyv-toy-consensus-epr-8-3 | 3 | 1 | 1 | 65,543 | 1m1s | 70,903 | **6s** | 70,903 | 7s | S4 | 70,903 | **2s** |
| ex-toy-consensus-8-3 | 2 | 1 | 1 | 65,543 | 57s | 70,903 | **5s** | 70,903 | 6s | S4 | 70,903 | **2s** |
| pyv-client-server-db-ae-2-3-2 | 5 | 4 | 4 | 188,158 | **12s** | 1,394,368 | 1m1s | 188,799 | **15s** | S1 | 1,394,368 | 18s |
| pyv-client-server-db-ae-4-2-2 | 5 | 1 | 1 | 356,706 | **1m23s** | 3,624,960 | 2m48s | 356,706 | 1m40s | S1 | 3,624,960 | **44s** |
| pyv-firewall-5 | 2 | 0 | 0 | 56,072 | 9s | 56,072 | **2s** | 56,072 | 3s | S4 | 56,072 | **1s** |
| ex-majorityset-leader-election-5 | 3 | 1 | - | - | TO | 166,306 | **15s** | 166,306 | 17s | S4 | 166,306 | **5s** |
| pyv-consensus-epr-4-4 | 6 | 2 | 1 | 7,018 | **3s** | - | TO | 7,221 | **5s** | S3 | - | TO |
| mldr-2 | 1 | 0 | - | - | TO | - | TO | - | TO | - | - | TO |

Fig. 6: Run time comparison between Recomp-Verify and TLC. The superscripts for each tool indicates how many threads are allocated to a trial. The fastest times for each experiment are bolded. The "Strat." column denotes the fastest strategy.

by two research questions. First, (**RQ1**) can hand-written recomposition maps provide more efficient verification than TLC? If this is the case, we then ask whether our technique is still performant when automating the search for recomposition maps. More precisely, (**RQ2**) is the performance of Recomp-Verify (using a parallel, portfolio strategy) competitive with TLC when each tool is allotted four threads?

In our experiments, we use TLC$^1$ and TLC$^4$ to respectively denote TLC run with one and four parallel threads; this is a built-in option for the tool. Recomp-Verify$^1$ is the version of our tool with one thread and hand-crafted maps, while Recomp-Verify$^4$ denotes the version that uses four threads to run the portfolio of recomposition strategies (S1-4) from Sec. VI in parallel. We report the fastest strategy for Recomp-Verify$^4$ in the "Strat." column in Fig. 6. Additionally, in the implementation of Recomp-Verify$^4$, we use TLC$^1$ for running the monolithic strategy (S4), since TLC is far more efficient than our research prototype for monolithic model checking. For example, in ex-quorum-leader-election-6 in Fig. 6, Recomp-Verify$^1$ uses an optimal single-threaded strategy, yet is slower than TLC$^1$–and therefore Recomp-Verify$^4$ too.

Every experiment in this paper was run on an Apple MacBook Pro with 32GB of memory and an M1 processor. For each benchmark, we report the total run time using the Unix *time* utility as well as the maximum number of states checked. We use TO to indicate a timeout after ten minutes and OM to indicate a program crash due to reaching the memory limit given a 25GB allotment. For TLC's maximum state count, we use the number of unique states that the tool reports. For Recomp-Verify, we use the maximum between (1) the number of unique states generated for each component and, for each iteration, (2) the number of states that results from composition in Alg. 3 on line 5. For Recomp-Verify$^1$, we also report the number of components that result from decomposition ($n$), the number of recomposed components ($m$), and the number of recomposed components that were checked ($k$).

### C. Results and Discussion

*1) RQ1:* We show our results in Fig. 6. In terms of state space, TLC$^1$ enumerates at least as many states as Recomp-Verify$^1$ in every case. For six of the benchmarks that both tools verified, recomposition reduced the state size by *millions* of states. Moreover, Recomp-Verify$^1$ short-circuits ($k < m$) for eight benchmarks, each of which has a significantly smaller state space than TLC$^1$. Finally, Recomp-Verify$^1$–but not TLC$^1$–was able to verify the one infinite state benchmark (tla-twophase-counter) via static specification reduction.

In terms of verification speed, Recomp-Verify$^1$ and TLC$^1$ both outperform each other in fourteen benchmarks, and tie in five cases. However, Recomp-Verify$^1$ completes more

benchmarks, verifying twenty-nine benchmarks while TLC[1] verifies twenty-four. Generally speaking, Recomp-Verify[1] is more performant on larger benchmarks; on benchmarks with over a million states, TLC[1] is faster in two cases while Recomp-Verify[1] is faster in at least six cases. We therefore answer RQ1 by concluding that hand-crafted maps *can* provide more efficiency than TLC.

*2) RQ2:* The results for TLC[4] and Recomp-Verify[4] are similar to the single threaded versions. In terms of state space, Recomp-Verify[4] always enumerates the same number of (or fewer) states than TLC[4], and also exhibits large savings in the millions for six benchmarks. We note that Recomp-Verify[4] short-circuited every time that Recomp-Verify[1] short-circuited, which showcases the effectiveness of the data flow heuristic from Sec. VI-A.

In terms of verification speed, Recomp-Verify[4] is faster for eleven benchmarks, TLC[4] is faster for fourteen, and the tools tie for eight benchmarks. However, Recomp-Verify[4] verifies more benchmarks, completing thirty-two, while TLC[4] completes twenty-four. While threading made TLC faster for the smaller benchmarks, it did not help the tool verify more benchmarks. On the other hand, threading helped Recomp-Verify to verify three more benchmarks. Generally speaking, Recomp-Verify[4] outperforms TLC[4] for large benchmarks and is competitive with TLC[4] for smaller ones, therefore we answer RQ2 in the affirmative.

*3) Discussion:* Ultimately, Recomp-Verify tends to be faster than TLC on benchmarks that have more opportunity for recomposition. We point out that Recomp-Verify[1] is faster than TLC[1] in *every* case where decomposition produced at least four components ($n \geq 4$). The same is true for Recomp-Verify[4] and TLC[4] in all but one case. This observation suggests that the potential benefits of recomposition increase with the number of available components ($n$).

## VIII. Related Work

Compositional verification is a well studied research area. Two widely studied styles of compositional verification are CRA [4], [5], [6], [8], [31], [12], [13], [14], [15], [16] and assume-guarantee reasoning [2], [3], [7], [17], [9], [32], [18], [19], [10], [11], [33], although other styles exist as well [34], [35]. Of these works, the ones most closely related to this paper automate decomposition for verification. Metzler et a. [18] and Cobleigh et al. [17] decompose systems into two components, after which they apply $L^*$ style learning [36] to infer assumptions for assume-guarantee style compositional verification. Nam et al. [19] use a similar strategy, but consider multi-way decomposition and verification. While these works report limited success, we are able to find efficient verification problems via recomposition.

Our work also relates to program slicing [20], [21] and cone of influence reduction [37], both of which are techniques for static specification reduction. These two techniques soundly reduce the state variables needed for model checking by analyzing a variable dependency graph. Our work includes static specification reduction by allowing partial recomposition maps, as described in Sec. V.

In the TLA$^+$ ecosystem, TLC [23] is the most well-known model checker. Apalache [38], [39] is an alternate model checker that internally relies on SMT solvers. Apalache supports bounded model checking and verification with inductive invariants–two techniques that are outside the scope of comparison for our tool. The TLA$^+$ Proof System (TLAPS) [40] provides an alternative to model checking TLA$^+$. TLAPS proofs are manually constructed, but automatically verified by dispatching proof obligations to SMT solvers. Endive [30] is a tool that automatically infers inductive invariants for TLA$^+$ specifications, which then may be checked using a TLAPS proof.

## IX. Limitations and Future Work

In Sec. VII-C, we show that the effectiveness of our approach is tied to the number of $C_i$ components. In future work, we plan to investigate methods to make decomposition more granular, as well as decomposing *properties*. As the number of components increase, we also plan to improve our methods for finding efficient recomposition maps. For example, we plan to improve our parallel technique so different threads can share intermediate work to save time and memory; this may allow us to explore even more recomposition maps at once.

In this paper, we focus on using recomposition for explicit-state model checking; however, recomposition may also be effective for other compositional techniques. For example, we plan to investigate whether recomposition can be used in combination with non-explicit verification techniques, e.g. using SMT solvers or inductive invariants. We also plan to investigate whether recomposition can be used for efficient counter-example detection.


## Acknowledgements

The authors are most grateful to Changjian Zhang, Andy Hammer, and the anonymous reviewers for their helpful comments on earlier versions of this paper. This project was supported by the U.S. NSF Award #2144860.


## References


[1] D. Giannakopoulou, K. S. Namjoshi, and C. S. Păsăreanu, *Compositional Reasoning*. Cham: Springer International Publishing, 2018, pp. 345–383. [Online]. Available: https://doi.org/10.1007/978-3-319-10575-8_12

[2] R. Alur, P. Madhusudan, and W. Nam, "Symbolic compositional verification by learning assumptions," in *Computer Aided Verification*, K. Etessami and S. K. Rajamani, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 548–562.

[3] Y.-F. Chen, A. Farzan, E. M. Clarke, Y.-K. Tsay, and B.-Y. Wang, "Learning minimal separating dfa's for compositional verification," in *Tools and Algorithms for the Construction and Analysis of Systems*, S. Kowalewski and A. Philippou, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 31–45.

[4] S. C. Cheung and J. Kramer, "Compositional reachability analysis of finite-state distributed systems with user-specified constraints," in *Proceedings of the 3rd ACM SIGSOFT Symposium on Foundations of Software Engineering*, ser. SIGSOFT '95. New York, NY, USA: Association for Computing Machinery, 1995, p. 140–150. [Online]. Available: https://doi.org/10.1145/222124.222149

[5] ——, "Context constraints for compositional reachability analysis," *ACM Trans. Softw. Eng. Methodol.*, vol. 5, no. 4, p. 334–377, oct 1996. [Online]. Available: https://doi.org/10.1145/235321.235323

[6] ——, "Checking safety properties using compositional reachability analysis," *ACM Trans. Softw. Eng. Methodol.*, vol. 8, no. 1, p. 49–78, jan 1999. [Online]. Available: https://doi.org/10.1145/295558.295570

[7] J. M. Cobleigh, D. Giannakopoulou, and C. S. PĂsĂreanu, "Learning assumptions for compositional verification," in *Tools and Algorithms for the Construction and Analysis of Systems*, H. Garavel and J. Hatcliff, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 331–346.

[8] S. Graf and B. Steffen, "Compositional minimization of finite state systems," in *Proceedings of the 2nd International Workshop on Computer Aided Verification*, ser. CAV '90. Berlin, Heidelberg: Springer-Verlag, 1990, p. 186–196.

[9] A. Gupta, K. L. McMillan, and Z. Fu, "Automated assumption generation for compositional verification," in *Proceedings of the 19th International Conference on Computer Aided Verification*, ser. CAV'07. Berlin, Heidelberg: Springer-Verlag, 2007, p. 420–432.

[10] C. Păsăreanu, D. Giannakopoulou, M. Bobaru, J. Cobleigh, and H. Barringer, "Learning to divide and conquer: Applying the l*algorithm to automate assume-guarantee reasoning," *Formal Methods in System Design*, vol. 32, pp. 175–205, 06 2008.

[11] C. S. Păsăreanu and D. Giannakopoulou, "Towards a compositional spin," in *Model Checking Software*, A. Valmari, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 234–251.

[12] K. Sabnani, A. Lapone, and M. Uyar, "An algorithmic procedure for checking safety properties of protocols," *IEEE Transactions on Communications*, vol. 37, no. 9, pp. 940–948, 1989.

[13] K.-C. Tai and P. Koppol, "Hierarchy-based incremental analysis of communication protocols," in *1993 International Conference on Network Protocols*, 1993, pp. 318–325.

[14] H. Zheng, "Compositional reachability analysis for efficient modular verification of asynchronous designs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 29, no. 3, pp. 329–340, 2010.

[15] ——, "Local state space construction for compositional verification of concurrent systems," in *Proceedings of the 2014 International SPIN Symposium on Model Checking of Software*, ser. SPIN 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 11–19. [Online]. Available: https://doi.org/10.1145/2632362.2632366

[16] H. Zheng, Z. Zhang, C. J. Myers, E. Rodriguez, and Y. Zhang, "Compositional model checking of concurrent systems," *IEEE Transactions on Computers*, vol. 64, no. 6, pp. 1607–1621, 2015.

[17] J. M. Cobleigh, G. S. Avrunin, and L. A. Clarke, "Breaking up is hard to do: an investigation of decomposition for assume-guarantee reasoning," in *Proceedings of the 2006 International Symposium on Software Testing and Analysis*, ser. ISSTA '06. New York, NY, USA: Association for Computing Machinery, 2006, p. 97–108. [Online]. Available: https://doi.org/10.1145/1146238.1146250

[18] B. Metzler, H. Wehrheim, and D. Wonisch, "Decomposition for compositional verification," in *Formal Methods and Software Engineering*, S. Liu, T. Maibaum, and K. Araki, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 105–125.

[19] W. Nam, P. Madhusudan, and R. Alur, "Automatic symbolic compositional verification by learning assumptions," *Form. Methods Syst. Des.*, vol. 32, no. 3, p. 207–234, jun 2008. [Online]. Available: https://doi.org/10.1007/s10703-008-0055-8

[20] I. Brückner and H. Wehrheim, "Slicing an integrated formal method for verification," in *Formal Methods and Software Engineering*, K.-K. Lau and R. Banach, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 360–374.

[21] M. Weiser, "Programmers use slices when debugging," *Commun. ACM*, vol. 25, no. 7, p. 446–452, jul 1982. [Online]. Available: https://doi.org/10.1145/358557.358577

[22] L. Lamport, *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, June 2002. [Online]. Available: https://www.microsoft.com/en-us/research/publication/specifying-systems-the-tla-language-and-tools-for-hardware-and-software-engineers/

[23] Y. Yu, P. Manolios, and L. Lamport, "Model checking tla+ specifications," in *Correct Hardware Design and Verification Methods*, L. Pierre and T. Kropf, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 54–66.

[24] J. Gray and L. Lamport, "Consensus on transaction commit," *ACM Trans. Database Syst.*, vol. 31, no. 1, p. 133–160, mar 2006. [Online]. Available: https://doi.org/10.1145/1132863.1132867

[25] D. Giannakopoulou, C. Pasareanu, and H. Barringer, "Assumption generation for software component verification," in *Proceedings 17th IEEE International Conference on Automated Software Engineering,*, 2002, pp. 3–12.

[26] C. A. R. Hoare, "Communicating sequential processes," *Commun. ACM*, vol. 21, no. 8, p. 666–677, aug 1978. [Online]. Available: https://doi.org/10.1145/359576.359585

[27] W. J. Yeh and M. Young, "Compositional reachability analysis using process algebra," in *Proceedings of the Symposium on Testing, Analysis, and Verification*, ser. TAV4. New York, NY, USA: Association for Computing Machinery, 1991, p. 49–59. [Online]. Available: https://doi.org/10.1145/120807.120812

[28] R. Milner, *Communication and Concurrency*, ser. Ph/AMA Series in Marketing. Prentice Hall, 1989. [Online]. Available: https://books.google.com/books?id=S5UZAQAAIAAJ

[29] "Recomp-verify research prototype model checker for tla+," https://github.com/cmu-soda/recomp-verify/tree/FMCAD24, accessed: 2024-08-16.

[30] W. Schultz, I. Dardik, and S. Tripakis, "Plain and simple inductive invariant inference for distributed protocols in tla+," in *2022 Formal Methods in Computer-Aided Design (FMCAD)*. IEEE, 2022, pp. 273–283.

[31] J. Malhotra, S. A. Smolka, A. Giacalone, and R. Shapiro, "Winston: A tool for hierarchical design and simulation of concurrent systems," in *Specification and Verification of Concurrent Systems*, C. Rattray, Ed. London: Springer London, 1990, pp. 140–152.

[32] C. Jones, "Specification and design of (parallel) programs." vol. 83, 01 1983, pp. 321–332.

[33] A. Pnueli, "In transition from global to modular temporal reasoning about programs," in *Logics and Models of Concurrent Systems*, K. R. Apt, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1985, pp. 123–144.

[34] H. Barringer, D. Giannakopoulou, and C. Pasareanu, "Proof rules for automated compositional verification through learning," 02 2003.

[35] S. Bensalem, M. Bozga, J. Sifakis, and T.-H. Nguyen, "Compositional verification for component-based systems and application," in *Automated Technology for Verification and Analysis*, S. S. Cha, J.-Y. Choi, M. Kim, I. Lee, and M. Viswanathan, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 64–79.

[36] D. Angluin, "Learning regular sets from queries and counterexamples," *Inf. Comput.*, vol. 75, no. 2, p. 87–106, nov 1987. [Online]. Available: https://doi.org/10.1016/0890-5401(87)90052-6

[37] E. Clarke, O. Grumberg, D. Kroening, D. Peled, and H. Veith, *Model Checking, second edition*, ser. Cyber Physical Systems Series. MIT Press, 2018. [Online]. Available: https://books.google.com/books?id=qJl8DwAAQBAJ

[38] I. Konnov, J. Kukovec, and T.-H. Tran, "Tla+ model checking made symbolic," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, oct 2019. [Online]. Available: https://doi.org/10.1145/3360549

[39] R. Otoni, I. Konnov, J. Kukovec, P. Eugster, and N. Sharygina, "Symbolic model checking for tla+ made faster," in *Tools and Algorithms for the Construction and Analysis of Systems*, S. Sankaranarayanan and N. Sharygina, Eds. Cham: Springer Nature Switzerland, 2023, pp. 126–144.

[40] D. Cousineau, D. Doligez, L. Lamport, S. Merz, D. Ricketts, and H. Vanzetto, "Tla+ proofs," *Proceedings of the 18th International Symposium on Formal Methods (FM 2012), Dimitra Giannakopoulou and Dominique Mery, editors. Springer-Verlag Lecture Notes in Computer Science*, vol. 7436, pp. 147–154, January 2012. [Online]. Available: https://www.microsoft.com/en-us/research/publication/tla-proofs/

## APPENDIX A
## PROOF OF THM. 1

Before proving Thm 1, we first introduce more general semantics for TLA$^+$ specifications, namely action-state-based semantics. We define an action-state-based behavior $\sigma$ as an infinite sequence of action-state pairs: $(\epsilon, b_0)(a_1, b_1)(a_2, b_2)\ldots$, where $\epsilon$ is the empty action. We use the notation $\sigma_i$ to access

the $i$th pair of $\sigma$. Notice that we require the initial action to be empty, hence $\sigma_0 = (\epsilon, b_0)$. Let $\phi$ be an arbitrary TLA$^+$ formula, let $\psi$ be a TLA$^+$ formula without temporal operators, and let $Next$ be the transition relation for a specification $S$, i.e. $Next$ is a TLA$^+$ formula with prime operators. We now define the action-state-based semantics of a behavior $\sigma$ on a TLA$^+$ formula in terms of the action-state-based satisfaction operator $\vdash$.

$$\begin{array}{lll}
\sigma \vdash \psi & \text{iff} & b_0 \models \psi \\
\sigma \vdash \Box\phi & \text{iff} & \forall i \in \mathbb{N} \ : \ \sigma_i\sigma_{i+1} \ldots \vdash \phi \\
\sigma \vdash \Diamond\phi & \text{iff} & \exists i \in \mathbb{N} \ : \ \sigma_i\sigma_{i+1} \ldots \vdash \phi \\
\sigma \vdash [Next]_v & \text{iff} & \big(a_1 \in \alpha S \wedge b'_1 \models (b_0 \wedge a_1)\big) \vee \\
 & & \big(a_1 \notin \alpha S \wedge b'_1 \models (b_0 \wedge v' = v)\big)
\end{array}$$

Note that the semantics for $\sigma \vdash [Next]_v$ implicitly uses $Next$ because $\alpha S$ is defined by $Next$. We now define the action-state-based semantics of a TLA$^+$ specification $S$ as the set of all action-state-based behaviors that satisfy $S!Spec$. More formally, we define the action-state-based semantics of $S$ to be: $[\![S]\!]_{\alpha\beta} = \{\sigma \mid \sigma \vdash S!Spec\}$. We also define $actions(\sigma)$ to denote the action-based behavior $a_1\, a_2 \ldots$. We overload this operator on a set of action-state-based behaviors $A$ as follows: $actions(A) = \{actions(\sigma) \mid \sigma \in A\}$.

**Lemma 3.** $[\![S /\!/ T]\!]_{\alpha\beta} = [\![S]\!]_{\alpha\beta} \cap [\![T]\!]_{\alpha\beta}$

*Proof.* We will first show that $[\![S /\!/ T]\!]_{\alpha\beta} \subseteq [\![S]\!]_{\alpha\beta} \cap [\![T]\!]_{\alpha\beta}$. Suppose that $\sigma \in [\![S /\!/ T]\!]_{\alpha\beta}$, then it suffices to show that $\sigma \vdash S!Spec$ and $\sigma \vdash T!Spec$. Recall that $Spec$ has the form $Init \wedge \Box[Next]_{vars}$. Hence, we break the proof obligation into two tasks, one for each conjunct. The first task is to show that $\sigma \vdash S!Init$ and $\sigma \vdash T!Init$; this fact follows because $\sigma \vdash (S!Init \wedge T!Init)$ by the definition of $/\!/$. The second task is to show that $\sigma \vdash \Box[S!Next]_{vars}$ and $\sigma \vdash \Box[T!Next]_{vars}$. By the definition of the action-state-based semantics for TLA$^+$, this is equivalent to showing (for just $S$):

$$\begin{array}{l}
\forall i \in \mathbb{N} \ : \ \vee \big(a_{i+1} \in \alpha S \wedge b'_{i+1} \models (b_i \wedge S!a_{i+1})\big) \\
\qquad\qquad \vee \big(a_{i+1} \notin \alpha S \wedge b'_{i+1} \models (b_i \wedge S!vars' = S!vars)\big)
\end{array} \tag{8}$$

We will prove equation (8)–for both $S$ and $T$–by considering the following four cases for an arbitrary choice of $i$:

1) Case 1: $a_{i+1} \in \alpha S$ and $a_{i+1} \in \alpha T$.
   In this case, the symbolic action for $a_{i+1}$ must be in $\hat{\alpha}S$ and $\hat{\alpha}T$, and therefore $(S /\!/ T)!a_{i+1}$ is equal to $S!a_{i+1} \wedge T!a_{i+1}$. Thus we see that $b'_{i+1} \models b_i \wedge S!a_{i+1} \wedge T!a_{i+1}$ by the action-state-based semantics of TLA$^+$. This implies equation (8) for the base case for both $S$ and $T$.
2) Case 2: $a_{i+1} \in \alpha S$ and $a_{i+1} \notin \alpha T$.
   In this case, the symbolic action for $a_{i+1}$ must be in $\hat{\alpha}S$ but not in $\hat{\alpha}T$. Therefore, $(S /\!/ T)!a_{i+1}$ is equal to $S!a_{i+1} \wedge (T!vars' = T!vars)$. Thus we see that $b'_{i+1} \models b_i \wedge S!a_{i+1} \wedge (T!vars' = T!vars)$ by the action-state-based semantics of TLA$^+$. This implies equation (8) for the base case for both $S$ and $T$.
3) Case 3: $a_{i+1} \notin \alpha S$ and $a_{i+1} \in \alpha T$.
   This case is identical to Case 2, except with $S$ and $T$ flipped.
4) Case 4: $a_{i+1} \notin \alpha S$ and $a_{i+1} \notin \alpha T$.
   In this case, the symbolic action for $a_{i+1}$ must not be in $\hat{\alpha}S$ nor $\hat{\alpha}T$. Therefore, $(S /\!/ T)!a_{i+1}$ is equal to $(S!vars' = S!vars) \wedge (T!vars' = T!vars)$. Thus we see that $b'_{i+1} \models b_i \wedge (S!vars' = S!vars) \wedge (T!vars' = T!vars)$ by the action-state-based semantics of TLA$^+$. This implies equation (8) for the base case for both $S$ and $T$.

Conversely, we will now show that $[\![S]\!]_{\alpha\beta} \cap [\![T]\!]_{\alpha\beta} \subseteq [\![S /\!/ T]\!]_{\alpha\beta}$. Suppose that $\sigma \in [\![S]\!]_{\alpha\beta} \cap [\![T]\!]_{\alpha\beta}$, then it suffices to show that $\sigma \vdash (S /\!/ T)!Spec$. Recall that $Spec$ has the form $Init \wedge \Box[Next]_{vars}$. Hence, we break the proof obligation into two tasks, one for each conjunct. The first task is to show $\sigma \vdash (S/\!/T)!Init$, or equivalently $\sigma \vdash S!Init \wedge T!Init$. Because $\sigma \vdash S!Spec$ and $\sigma \vdash T!Spec$, we see that $\sigma \vdash S!Init$ and $\sigma \vdash T!Init$; thus the first task is proved. The second task is to show that $\sigma \vdash \Box[(S /\!/ T)!Next]_{vars}$. By the definition of the action-state-based semantics for TLA$^+$, this is equivalent to showing:

$$\begin{array}{l}
\forall i \in \mathbb{N} \ : \\
\ \vee \big(a_{i+1} \in \alpha(S /\!/ T) \wedge b'_{i+1} \models (b_i \wedge (S /\!/ T)!a_{i+1})\big) \\
\ \vee \big(a_{i+1} \notin \alpha(S /\!/ T) \wedge b'_{i+1} \models \\
\qquad (b_i \wedge (S /\!/ T)!vars' = (S /\!/ T)!vars)\big)
\end{array} \tag{9}$$

We will prove equation (9) by considering the following four cases for an arbitrary choice of $i$:

1) Case 1: $a_{i+1} \in \alpha S$ and $a_{i+1} \in \alpha T$.
   In this case, $a_{i+1} \in \alpha(S /\!/ T)$. Next, notice that the symbolic action for $a_{i+1}$ must be in $\hat{\alpha}S$ and $\hat{\alpha}T$, and therefore $(S/\!/T)!a_{i+1}$ is equal to $S!a_{i+1} \wedge T!a_{i+1}$. Furthermore, $b'_{i+1} \models b_i \wedge S!a_{i+1}$ and $b'_{i+1} \models b_i \wedge T!a_{i+1}$. Therefore we conclude $b'_{i+1} \models b_i \wedge S!a_{i+1} \wedge T!a_{i+1}$ which implies equation (9) for this case.
2) Case 2: $a_{i+1} \in \alpha S$ and $a_{i+1} \notin \alpha T$.
   In this case, $a_{i+1} \in \alpha(S /\!/ T)$. Next, notice that the symbolic action for $a_{i+1}$ must be in $\hat{\alpha}S$ but not in $\hat{\alpha}T$. Therefore, $(S /\!/ T)!a_{i+1}$ is equal to $S!a_{i+1} \wedge (T!vars' = T!vars)$. Furthermore, $b'_{i+1} \models b_i \wedge S!a_{i+1}$ and $b'_{i+1} \models b_i \wedge (T!vars' = T!vars)$. Therefore we conclude $b'_{i+1} \models b_i \wedge S!a_{i+1} \wedge (T!vars' = T!vars)$ which implies equation (9) for this case.
3) Case 3: $a_{i+1} \notin \alpha S$ and $a_{i+1} \in \alpha T$.
   This case is identical to Case 2, except with $S$ and $T$ flipped.
4) Case 4: $a_{i+1} \notin \alpha S$ and $a_{i+1} \notin \alpha T$.
   In this case, $a_{i+1} \notin \alpha(S /\!/ T)$. Next, notice that the symbolic action for $a_{i+1}$ must not be in $\hat{\alpha}S$ nor $\hat{\alpha}T$. Therefore, $(S /\!/ T)!a_{i+1}$ is equal to $(S!vars' = S!vars) \wedge (T!vars' = T!vars)$. Furthermore, $b'_{i+1} \models b_i \wedge (S!vars' = S!vars)$ and $b'_{i+1} \models b_i \wedge (T!vars' = T!vars)$. Therefore we conclude $b'_{i+1} \models b_i \wedge ((S!vars \circ$

$T!vars)' = (S!vars \circ T!vars))$. Since $(S /\!\!/ T)!vars = (S!vars \circ T!vars)$, this implies equation (9) for this case. □

Lem. 3 shows that our notion of parallel composition is semantically an intersection in the action-state-based semantics of TLA$^+$. We now prove one more lemma before proving Thm. 1.

**Lemma 4.** *Let $\mathcal{A}$ and $\mathcal{B}$ be LTSs, then $[\![\mathcal{A} \parallel \mathcal{B}]\!]_\alpha = [\![\mathcal{A}]\!]_\alpha \cap [\![\mathcal{B}]\!]_\alpha$.*

*Proof.* We first prove that $[\![\mathcal{A} \parallel \mathcal{B}]\!]_\alpha \subseteq [\![\mathcal{A}]\!]_\alpha \cap [\![\mathcal{B}]\!]_\alpha$. Suppose that $\sigma \in [\![\mathcal{A} \parallel \mathcal{B}]\!]_\alpha$. Then there exists a sequence of state pairs $(p_0, q_0), (p_1, q_1), \ldots$ such that $p_0 \in I_\mathcal{A}$, $q_0 \in I_\mathcal{B}$ and, for each nonnegative index $i$, either (1) $\sigma_i \in \alpha\mathcal{A}$, $(p_i, \sigma_i, p_{i+1}) \in \delta_\mathcal{A}$, $\sigma_i \in \alpha\mathcal{B}$, and $(q_i, \sigma_i, q_{i+1}) \in \delta_\mathcal{B}$, (2) $\sigma_i \in \alpha\mathcal{A}$, $(p_i, \sigma_i, p_{i+1}) \in \delta_\mathcal{A}$, $\sigma_i \notin \alpha\mathcal{B}$, and $q_i = q_{i+1}$, (3) $\sigma_i \notin \alpha\mathcal{A}$, $p_i = p_{i+1}$, $\sigma_i \in \alpha\mathcal{B}$, and $(q_i, \sigma_i, q_{i+1}) \in \delta_\mathcal{B}$, or (4) $\sigma_i \notin \alpha\mathcal{A}$, $p_i = p_{i+1}$, $\sigma_i \notin \alpha\mathcal{B}$, and $q_i = q_{i+1}$. In each of these four cases, $\sigma_i$ is a valid action from $p_i$ to $p_{i+1}$ in $\mathcal{A}$ and also a valid action from $q_i$ to $q_{i+1}$ in $\mathcal{B}$. Given that $p_0 \in I_\mathcal{A}$ and $q_0 \in I_\mathcal{B}$, we have $\sigma \in [\![\mathcal{A}]\!]_\alpha$ and $\sigma \in [\![\mathcal{B}]\!]_\alpha$.

Conversely, we will now show that $[\![\mathcal{A}]\!]_\alpha \cap [\![\mathcal{B}]\!]_\alpha \subseteq [\![\mathcal{A} \parallel \mathcal{B}]\!]_\alpha$. Suppose that $\sigma \in [\![\mathcal{A}]\!]_\alpha \cap [\![\mathcal{B}]\!]_\alpha$. Then there exists a sequence of states $p_0, p_1, \ldots$ such that $p_0 \in I_\mathcal{A}$ and, for each nonnegative index $i$, either (1) $\sigma_i \in \alpha\mathcal{A}$ and $(p_i, \sigma_i, p_{i+1}) \in \delta$, or (2) $\sigma_i \notin \alpha\mathcal{A}$ and $p_i = p_{i+1}$. Likewise, there exists a sequence of state $q_0, q_1, \ldots$ with the same conditions but for $\mathcal{B}$. Then we can construct a sequence of state pairs $(p_0, q_0), (p_1, q_1), \ldots$ such that $p_0 \in I_\mathcal{A}$, $q_0 \in I_\mathcal{B}$ and, for each nonnegative index $i$, at least one of the following four conditions holds: (1) $\sigma_i \in \alpha\mathcal{A}$, $(p_i, \sigma_i, p_{i+1}) \in \delta_\mathcal{A}$, $\sigma_i \in \alpha\mathcal{B}$, and $(q_i, \sigma_i, q_{i+1}) \in \delta_\mathcal{B}$, (2) $\sigma_i \in \alpha\mathcal{A}$, $(p_i, \sigma_i, p_{i+1}) \in \delta_\mathcal{A}$, $\sigma_i \notin \alpha\mathcal{B}$, and $q_i = q_{i+1}$, (3) $\sigma_i \notin \alpha\mathcal{A}$, $p_i = p_{i+1}$, $\sigma_i \in \alpha\mathcal{B}$, and $(q_i, \sigma_i, q_{i+1}) \in \delta_\mathcal{B}$, or (4) $\sigma_i \notin \alpha\mathcal{A}$, $p_i = p_{i+1}$, $\sigma_i \notin \alpha\mathcal{B}$, and $q_i = q_{i+1}$. Thus, by definition, $\sigma \in [\![\mathcal{A} \parallel \mathcal{B}]\!]_\alpha$. □

Using the prior two lemmas, we now prove Thm. 1.

**Theorem 1.** $[\![\text{LTS}(S /\!\!/ T)]\!]_\alpha = [\![\text{LTS}(S) \parallel \text{LTS}(T)]\!]_\alpha$.

*Proof.*

$$\begin{aligned}
[\![\text{LTS}(S /\!\!/ T)]\!]_\alpha &= \mathit{actions}\,([\![S /\!\!/ T]\!]_{\alpha\beta}) \\
&= \mathit{actions}\,([\![S]\!]_{\alpha\beta} \cap [\![T]\!]_{\alpha\beta}) \\
&= \mathit{actions}\,([\![S]\!]_{\alpha\beta}) \cap \mathit{actions}\,([\![T]\!]_{\alpha\beta}) \\
&= [\![\text{LTS}(S)]\!]_\alpha \cap [\![\text{LTS}(T)]\!]_\alpha \\
&= [\![\text{LTS}(S) \parallel \text{LTS}(T)]\!]_\alpha
\end{aligned}$$

Where the second equality is due to Lem. 3, the third equality holds because $S$ and $T$ do not share state variables, and the fifth equality is due to Lem. 4. □

## Appendix B
## Static Specification Reduction

We begin by showing that the sequence $\{X_i\}$ converges by the $n+1^{\text{st}}$ term, i.e. at $X_n$.

**Theorem 4.** $X_n = X_{n+1}$

*Proof.* First, we note that $n + 1 > |X_{n+1}|$ because there are at most $n$ components that can be in each $X_i$. Next, a simple inductive argument shows that whenever $X_i \neq X_{i+1}$, $|X_{i+1}| \geq i + 1$. Finally, for the sake of contradiction, suppose that $X_n \neq X_{n+1}$. Then it would be the case that $n + 1 > |X_{n+1}| \geq n + 1$ which is a contradiction. □

Before presenting Thm. 2, we define $\mathit{states}(\sigma)$ on an action-state-based behavior $\sigma$ to denote the state-based behavior $b_0 b_1 \ldots$. We overload this operator on a set of action-state-based behaviors $B$ as follows: $\mathit{states}(B) = \{\mathit{states}(\sigma) \mid \sigma \in B\}$.

**Theorem 2.** *$S \models P$ if and only if $/\!\!/(\bigcup_i X_i) \models P$.*

*Proof.* Let $N = /\!\!/(\bigcup_i X_i)$ be the composition of the necessary components and let $U = /\!\!/(\{C_1, \ldots, C_n\} - \bigcup_i X_i)$ be the composition of the unnecessary components. We note that $\hat{\alpha}N \cap \hat{\alpha}U = \emptyset$, which follows from the definition of $X_{i+1}$. Also, because $N$ is composed of $C_1$, $\beta P \subseteq \beta N$ by property (P2) of Lem. 1. Because $\beta P \subseteq \beta N$ and components do not share state variables, we can therefore conclude that $\beta P \cap \beta U = \emptyset$. Now we have:

$$\begin{aligned}
S \models P &\iff N /\!\!/ U \models P \\
&\iff \mathit{states}([\![N /\!\!/ U]\!]_{\alpha\beta}) \subseteq [\![P]\!]_\beta \\
&\iff \mathit{states}([\![N]\!]_{\alpha\beta} \cap [\![U]\!]_{\alpha\beta}) \subseteq [\![P]\!]_\beta \\
&\iff \mathit{states}([\![N]\!]_{\alpha\beta}) \subseteq [\![P]\!]_\beta \\
&\iff N \models P
\end{aligned}$$

In the equation above, the third biconditional is due to Lem. 3. The fourth biconditional follows for two reasons. The first reason is that $\beta P \cap \beta U = \emptyset$, and therefore the states in each behavior of $[\![U]\!]_{\alpha\beta}$ will not restrict the values of the variables in $\beta P$. The second reason is that $\hat{\alpha}N \cap \hat{\alpha}U = \emptyset$; this implies $\alpha N \cap \alpha U = \emptyset$, so the actions in each behavior of $[\![U]\!]_{\alpha\beta}$ will not restrict any behavior in $[\![N]\!]_{\alpha\beta}$. □

## Appendix C
## Slicing Procedure

We show the procedure for slicing in Fig. 7. Slicing works by creating a new specification $T$ such that $\beta T = V$. Line 2 in Fig. 7 defines $T$'s state variables to be $S!vars$ restricted to the variables in $V$, while line 3 defines $T$'s initial state predicate to be $S!Init$ restricted to the variables in $V$. Line 4 defines $acts$ to be the set of the actions in $S$ restricted to the variables in $V$, and line 5 defines $T$'s transition relation to be subset of $acts$ that are well-formed.

The Slice-Rec procedure on line 7 recursively descends into a TLA$^+$ formula to restrict it to a set of variables $V$. The procedure matches three cases (line 8): definitions (line 9), existential quantifiers (line 15), and conjunctions (line 21). We use the notation $\epsilon(.)$ to denote TLA$^+$ syntax, and $e_1$ (lines 9 and 15) and $c_i$ (line 21) both refer to arbitrary TLA$^+$ formulas. In a nutshell, slicing works by removing any conjunct that

1: **procedure** SLICE($S$, $V$)
2: $\quad T!vars \leftarrow \epsilon\big(S!vars \cap V\big)$
3: $\quad T!Init \leftarrow$ SLICE-REC($S!Init$, $V$)
4: $\quad acts \leftarrow \{$SLICE-REC($A$, $V$) $\mid A \in \hat{\alpha}S\}$
5: $\quad T!Next \leftarrow \epsilon\big(\bigvee\{A \in acts \mid A \neq \text{DEL}\}\big)$
6: $\quad$ **return** $T$
7: **procedure** SLICE-REC($e$, $V$)
8: $\quad$ **match** $e$ **with**
9: $\quad \epsilon\big(op(p) \triangleq e_1\big) \rightarrow$
10: $\quad\quad E \leftarrow$ SLICE-REC($e_1$, $V$)
11: $\quad\quad$ **if** $E = DEL$ **then**
12: $\quad\quad\quad$ **return** DEL
13: $\quad\quad$ **else**
14: $\quad\quad\quad$ **return** $\epsilon\big(op(p) \triangleq E\big)$

15: $\quad \epsilon\big(\exists x \in d \ : \ e_1\big) \rightarrow$
16: $\quad\quad E \leftarrow$ SLICE-REC($e_1$, $V$)
17: $\quad\quad$ **if** $E =$ DEL **then**
18: $\quad\quad\quad$ **return** DEL
19: $\quad\quad$ **else**
20: $\quad\quad\quad$ **return** $\epsilon\big(\exists x \in d \ : \ E\big)$
21: $\quad \epsilon\left(\bigwedge_{i \in I} c_i\right) \rightarrow$
22: $\quad\quad C \leftarrow \{c_j \mid \beta c_j \subseteq V\}$
23: $\quad\quad$ **if** $C = \emptyset$ **then**
24: $\quad\quad\quad$ **return** DEL
25: $\quad\quad$ **else**
26: $\quad\quad\quad$ **return** $\epsilon\,(\bigwedge C_j)$

Fig. 7: Slicing procedure definitions.

contains a variable not in $V$ (lines 22 and 26). In the case that *every* conjunct of a TLA$^+$ expression must be removed, we consider the entire expression malformed and we mark it for removal. We mark an expression for removal using the DEL keyword on line 24, which is then bubbled to the top (lines 12 and 18); the actual removal happens in the set comprehension on line 5.

## APPENDIX D
## THE DATA FLOW PARTIAL ORDER REFINES STATIC SPECIFICATION REDUCTION

The following theorem shows that the data flow partial order refines our static specification reduction scheme by providing *more* information. In other words, the data flow partial order identifies a class of components that are necessary for verification ($\bigcup_i X_i$) *and* provides an ordering over these components.

**Theorem 5.** *Let $G = \{C_i \mid \exists C_j : C_i \preccurlyeq C_j$ or $C_j \preccurlyeq C_i\}$ be the set of all specifications for which the data flow partial order is defined. Then $G = \bigcup_i X_i$, i.e. it is exactly the set of necessary components in static specification reduction.*

*Proof.* Based on the definition for $X_{i+1}$, we provide an alternate formulation for $E_{i+1}$:

$$\begin{aligned} E_{i+1} &= \{C_j \mid C_j \notin X_i \text{ and } \hat{\alpha}C_j \cap \hat{\alpha}E_i \neq \emptyset\} \\ &= \{C_j \mid C_j \notin X_i\} \cap \{C_j \mid \hat{\alpha}C_j \cap \hat{\alpha}E_i \neq \emptyset\} \end{aligned}$$

We now provide a formula for the set of all first (lesser) elements in each pair for $F_{i+1}$. We will refer to this set as $proj_1(F_{i+1})$ for the projection onto the first element.

$$\begin{aligned} proj_1(F_{i+1}) &= \{C_j \mid C_j \in E_i \text{ and} \\ &\qquad \exists C_k \in E_{i+1} : \hat{\alpha}C_j \cap \hat{\alpha}C_k \neq \emptyset\} \\ &= E_i \cap \{C_j \mid \hat{\alpha}C_j \cap \hat{\alpha}E_{i+1} \neq \emptyset\} \\ &= E_i \end{aligned}$$

Where the final equality is due to the alternate formulation of $E_{i+1}$.

We now provide a formula for the set of all second (greater) elements in each pair for $F_{i+1}$. We will refer to this set as $proj_2(F_{i+1})$ for the projection onto the second element.

$$\begin{aligned} proj_2(F_{i+1}) &= \{C_k \mid C_k \in E_{i+1} \text{ and} \\ &\qquad \exists C_j \in E_i : \hat{\alpha}C_j \cap \hat{\alpha}C_k \neq \emptyset\} \\ &= \{C_k \mid C_k \in E_{i+1} \text{ and } \hat{\alpha}E_i \cap \hat{\alpha}C_k \neq \emptyset\} \\ &= E_{i+1} \cap \{C_k \mid \hat{\alpha}E_i \cap \hat{\alpha}C_k \neq \emptyset\} \\ &= E_{i+1} \end{aligned}$$

Where the final equality is due to the alternate formulation of $E_{i+1}$. Putting it all together we see:

$$\begin{aligned} \bigcup_{i\geq 0} X_i = \bigcup_{i\geq 0} E_i &= \left(\bigcup_{i\geq 0} E_i\right) \cup \left(\bigcup_{i\geq 0} E_{i+1}\right) \\ &= \left(\bigcup_{i\geq 0} proj_1(F_{i+1})\right) \cup \left(\bigcup_{i\geq 0} proj_2(F_{i+1})\right) \\ &= \left(\bigcup_{i\geq 0} proj_1(F_i)\right) \cup \left(\bigcup_{i\geq 0} proj_2(F_i)\right) = G \end{aligned}$$

Where the penultimate equality follows because $F_0 = \emptyset$. □