Formal Methods in Computer-Aided Design 2024

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 [15].

Most research papers on compositional verification assume that the components are pre-determined and focus solely on verification algorithms [1, 6, 7, 9, 8, 12, 16, 18, 28, 29, 32, 34, 38, 39, 40]. 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 [11, 24, 26]. Unfortunately, research into automated decomposition has seen limited success thus far; as Cobleigh et al. lament, decomposing a system is tough [11].

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 $C_1, \dots, C_n$. 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 $D_1, \dots, D_m$ 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.

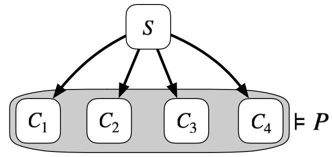
Additionally, we will show that our method is conducive to *specification reduction*. Specification reduction techniques, e.g. program slicing [5, 35], are generally considered separately from compositional verification. However, model checking with recomposition unites these two techniques under a single 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.

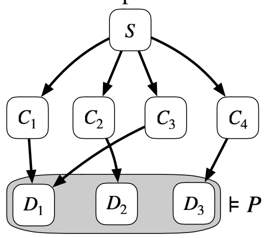
We have implemented our techniques in a model checker called "Recomp-Verify" for the TLA⁺ language [22]. We compare Recomp-Verify to TLC [37], a well-known model checker for TLA⁺. We show that recomposition can lead to large savings in terms of verification time and the size of the explored state space.

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, and (3) a prototype model checker Recomp-Verify that implements our algorithm, along with an evaluation of Recomp-Verify against TLC on a benchmark of distributed protocols.

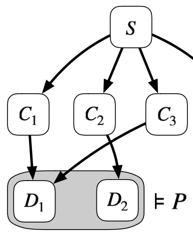
https://doi.org/



(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.

II. Motivating Example

In this section, we describe the Two Phase Commit Protocol [17] to motivate our work and serve as a running example throughout the paper.

a) 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 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.

b) 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:

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

c) 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.

d) 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

MODULE RM

VARIABLES $rmState$

$Init \triangleq$
  $\land\ rmState = [rm \in RMs \mapsto \text{“working”}]$

$SndPrepare(rm) \triangleq$
  $\land\ rmState[rm] = \text{“working”}$
  $\land\ rmState' = [rmState \text{ EXCEPT } ![rm] = \text{“prepared”}]$

MODULE Env

VARIABLES $msgs$

$Init \triangleq msgs = \{\}$

$SndPrepare(rm) \triangleq$
  $\land\ msgs' = msgs \cup \{[type \mapsto \text{“Prepared”},\ theRM \mapsto rm]\}$

$RcvPrepare(rm) \triangleq$
  $\land\ [type \mapsto \text{“Prepared”}, theRM \mapsto rm] \in msgs$
  $\land$ UNCHANGED $\langle msgs \rangle$

MODULE TM_1

VARIABLES $tmState$

$Init \triangleq tmState = \text{“init”}$

$RcvPrepare(rm) \triangleq$
  $\land\ tmState = \text{“init”}$
  $\land$ UNCHANGED $\langle tmState \rangle$

MODULE TM_2

VARIABLES $tmPrepared$

$Init \triangleq tmPrepared = \{\}$

$RcvPrepare(rm) \triangleq$
  $\land\ tmPrepared' = tmPrepared \cup \{rm\}$

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

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. 1, 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. VI, 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.

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

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

We define an action-based behavior σ as an infinite sequence of actions, i.e. $\sigma \in \mathbb{A}^\omega$, and we let σ_i denote the ith action in σ. 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 TLA+ 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 π state–that acts as a sink for any action that causes a safety violation. The second method is defining a *property LTS* whose language defines the safe behaviors; property LTSs must be deterministic and must not include a π state. Any error LTS can be converted to a property LTS using the construction described in [14], i.e. 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 [19] 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 $next$, 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 $Next \lor (vars' = vars)$ 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. We define the scoping operator ! for referencing definitions in S, e.g. $TP!SndPrepare$ refers to the $SndPrepare$ action of TP in Fig. 2. We use $\hat{\alpha}$ and α to refer to *symbolic actions* and *concrete actions* respectively. Symbolic actions are the action names in a specification, while concrete actions are the actions that may occur during an execution of a finite instance of a specification. 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 $\alpha\, TP_1 = \{SndPrepare(\text{“}rm1\text{”}), RcvPrepare(\text{“}rm1\text{”})\}$. Finally, we let βS denote the set of state variables in a specification or an expression, e.g. $\beta\, TP = \{msgs, rmState, tmState, tmPrepared\}$.

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 αS 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 π 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.

Parallel composition. In this paper, we use two notions of parallel composition. The first is the usual parallel composition operator $\parallel$ 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 $/\!\!/$. For example, the components in Fig. 3 constitute a decomposition of TP, i.e. $TP = RM \,/\!\!/\, Env \,/\!\!/\, TM_1 \,/\!\!/\, TM_2$. For the sake of space, we include a precise definition for the parallel composition operator in Appendix A. Additionally, we will use the notation $/\!\!/ Z$ to denote the composition over a set of specifications Z.

C. CRA-Style Compositional Verification

In this paper, we consider *compositional reachability analysis* (CRA) [7, 16, 36] 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 [8].

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 π 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 [25] 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. Model Checking with Recomposition

A. 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 1 (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. 1, 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.

B. The Recomp-Verify Algorithm

We present our model checking algorithm in Alg. 1. Alg. 1 accepts several inputs, including a *recomposition strategy*. A recomposition strategy ρ 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 ρ is given; we discuss recomposition strategy selection in Sec. V.

Alg. 1 begins by decomposing S into components on line 1; we provide more detail for decomposition in Sec. IV-C. 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

Algorithm 1 RECOMP-VERIFY

Input: Specification S, property P, recomposition strategy ρ
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: $D_P \leftarrow /\!\!/ f^{-1}(d_P)$ $\qquad \triangleright\ f^{-1}(d_P) \subseteq \{C_1, \ldots, C_n\}$
4: **for** $j \in \{1, \ldots, m\}$ **do**
5: $\quad D_j \leftarrow /\!\!/ f^{-1}(d_j)$ $\qquad \triangleright\ f^{-1}(d_j) \subseteq \{C_2, \ldots, C_n\}$
6: **return** $\text{COMP-VERIFY}(D_1, \ldots, D_m, D_P, P)$

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. IV-D.

Example 1. In this example we analyze Alg. 1 given the input TP, $Consistent$, and a hand-crafted optimal recomposition strategy ρ_{opt}. Line 1 of Alg. 1 will produce the components RM, Env, TM_1, and TM_2 from Fig. 3. ρ_{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 ρ_{opt} in this example reduces the maximum state space by 1027 states and also improves the model checking time from 1 minute 32 seconds to 51 seconds.

Recomposition maps that are total include every component during verification, hence checking the entire specification S. However, we are able to statically detect a class of components that are unnecessary for verification. We therefore allow recomposition maps to be partial, defined only for components that are necessary for verification. Partial recomposition maps are a *static specification reduction* technique, akin to program slicing [5, 35]. Due to space limitations, we expand upon static specification reduction in Appendix B.

C. Decomposition

In this section, we present an algorithm for decomposing a symbolic specification S into n components $C_1 \ldots C_n$. 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 Appendix C.

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 2. 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. IV-C2. Next, on lines 6-7, the algorithm slices TP into RM (Fig. 3) and a new specification T_1 (see Appendix C, Fig. 6). 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.

Algorithm 2 DECOMPOSE

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

1: $V_C, V_T \leftarrow \text{PARTITION}(S, \beta P)$
2: **if** $V_T = \emptyset$ **then**
3: $\quad$ **return** S
4: $T_0 \leftarrow S,\ i \leftarrow 0$
5: **while** $V_T \neq \emptyset$ **do**
6: $\quad C_{i+1} \leftarrow \text{SLICE}(T_i, V_C)$
7: $\quad T_{i+1} \leftarrow \text{SLICE}(T_i, V_T)$
8: $\quad v \in \beta T_{i+1}$ $\ \triangleright$ Nondeterministically choose a variable
9: $\quad V_C, V_T \leftarrow \text{PARTITION}(T_{i+1}, \{v\})$
10: $\quad i \leftarrow i + 1$
11: $n \leftarrow i + 1,\ C_n \leftarrow T_i$
12: **return** $C_1, \ldots, C_n$
13: **procedure** $\text{PARTITION}(T, V)$
14: $\quad V_C \leftarrow \text{FIX}(\text{OCCURS}_T, V)$
15: $\quad V_T \leftarrow \beta S - V_C$
16: $\quad$ **return** V_C, V_T
17: **procedure** $\text{OCCURS}_S(V)$
18: $\quad$ **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** $\text{FIX}(op, X)$
20: $\quad Y \leftarrow X \cup op(X)$
21: $\quad$ **if** X = Y **then return** X
22: $\quad$ **return** $Y \cup \text{FIX}(op, Y)$

2) State Variable Partitioning: Given a specification T_i, the partitioning phase partitions the variables βT_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 βP. 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 $\textsc{Fix}$ 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 3. 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 $V_T = \{tmState, tmPrepared\}$.

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:

$$\begin{aligned} Action \triangleq\ & \land\, var_1{}' = \text{“val1”} \\ & \land\, var_2{}' = \text{“val2”} \end{aligned}$$

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 $M_1 = \textsc{Slice}(M, \{var_1\})$, $M_2 = \textsc{Slice}(M, \{var_2\})$, and $M = M_1 /\!/ M_2$. In the TP example, this corresponds to $TP = RM /\!/ T_1$ in Ex. 2. We include more details on slicing, including the definition for the slicing procedure, in Appendix C.

D. Compositional Verification

We present a CRA-style compositional verification algorithm in Alg. 3; we provide a proof of correctness in Appendix D. The algorithm works by iteratively composing the LTS for each component D_j together (line 5) until the π state becomes unreachable, in which case verification succeeds (lines 3 and 7). If the π 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 [32]. 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 4. Consider TP with ten resource managers and the optimal mapping f from Ex. 1, 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 and a π state. Minimization reduces D_P to 13,291 states. Due to a reachable π state, the algorithm proceeds into the loop on line 4. Next, generating the LTS for D_1 has 3,072 states, which reduces to 1,026 states after minimization. Composing this LTS with $\mathcal{D}$ (line 5) retains the π state (line 6) so we loop again. The algorithm continues in this fashion until a π 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.

Algorithm 3 $\textsc{Comp-Verify}$

Input: $D_1, \ldots, D_m, D_P, P$
Output: If $\textsc{lts}(D_1) \parallel \ldots \parallel \textsc{lts}(D_m) \models \mathcal{P}$

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

E. Correctness Analysis

We rely on Thm. 1 to establish the soundness of Alg. 1. For sake of space, we include the proof in Appendix E.

Theorem 1. $S \models P \iff \textsc{lts}(D_1) \parallel \ldots \parallel \textsc{lts}(D_m) \models \mathcal{P}$.

While Thm. 1 shows that Alg. 1 is sound, the algorithm is not complete, even if we limit S to be a finite-state specification. This is because each component D_j is not guaranteed to be finite-state, and therefore LTS generation for each component in Alg. 3 may fail to terminate. We address this limitation in Sec. V-B by using a portfolio of strategies that includes the *monolithic strategy*.

V. 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. V-A. We then choose a small portfolio of strategies based on this heuristic in Sec. V-B. We provide an illustrative example in Appendix F, Ex. 6 for sake of space.

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 orders components based on how far removed their state variables are from impacting verification. We rigorously define the data flow partial order and prove it refines our static specification reduction scheme in Appendix F.

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. V-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.

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

VI. 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 [31].

B. Experiments

We evaluate Recomp-Verify against TLC on a benchmark of distributed protocols [33], plus the tla-twophase-counter protocol that we introduce in Appendix B. Our evaluation is driven 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. V in parallel. 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.

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.

C. Results and Discussion

We show our results in Fig. 5. 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.

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. V-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.

Ultimately, Recomp-Verify tends to be faster than TLC on benchmarks that have more opportunity for recomposition. We

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

Fig. 5: Run time comparison for Recomp-Verify and TLC. The superscripts for each tool indicates how many threads are allocated to a trial. Smallest state space and fastest times for each experiment are bolded.

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).

VII. Related Work

Compositional verification is a well studied research area. Two widely studied styles of compositional verification are CRA [7, 9, 8, 16, 23, 32, 34, 38, 39, 40] and assume-guarantee reasoning [1, 6, 12, 11, 18, 20, 24, 26, 28, 29, 30], although other styles exist as well [3, 4]. Of these works, the ones most closely related to this paper automate decomposition for verification. Metzler et a. [24] and Cobleigh et al. [11] decompose systems into two components, after which they apply L^* style learning [2] to infer assumptions for assume-guarantee style compositional verification. Nam et al. [26] 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 [5, 35] and cone of influence reduction [10], static 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. IV.

In the TLA$^+$ ecosystem, TLC [37] is the most well-known model checker. Apalache [21, 27] 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) [13] provides an alternative to model checking TLA$^+$. TLAPS proofs are manually constructed, but automatically verified by dispatching proof obligations to SMT solvers. Endive [33] is a tool that automatically infers inductive invariants for TLA$^+$ specifications, which then may be checked using a TLAPS proof.

VIII. Limitations and Future Work

In Sec. VI-C showed 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, allowing us to explore more recomposition maps at once.

References

[1] Rajeev Alur, P. Madhusudan, and Wonhong Nam. "Symbolic Compositional Verification by Learning Assumptions". In: *Computer Aided Verification*. Ed. by Kousha Etessami and Sriram K. Rajamani. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 548–562. ISBN: 978-3-540-31686-2.

[2] Dana Angluin. "Learning Regular Sets from Queries and Counterexamples". In: *Inf. Comput.* 75.2 (Nov. 1987), pp. 87–106. ISSN: 0890-5401. DOI: 10.1016/0890-5401(87)90052-6. URL: https://doi.org/10.1016/0890-5401(87)90052-6.

[3] Howard Barringer, Dimitra Giannakopoulou, and Corina Pasareanu. "Proof Rules for Automated Compositional Verification through Learning". In: (Feb. 2003).

[4] Saddek Bensalem et al. "Compositional Verification for Component-Based Systems and Application". In: *Automated Technology for Verification and Analysis*. Ed. by Sungdeok (Steve) Cha et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 64–79. ISBN: 978-3-540-88387-6.

[5] Ingo Brückner and Heike Wehrheim. "Slicing an Integrated Formal Method for Verification". In: *Formal Methods and Software Engineering*. Ed. by Kung-Kiu Lau and Richard Banach. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 360–374. ISBN: 978-3-540-32250-4.

[6] Yu-Fang Chen et al. "Learning Minimal Separating DFA's for Compositional Verification". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Stefan Kowalewski and Anna Philippou. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 31–45. ISBN: 978-3-642-00768-2.

[7] 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*. SIGSOFT '95. Washington, D.C., USA: Association for Computing Machinery, 1995, pp. 140–150. ISBN: 0897917162. DOI: 10.1145/222124.222149. URL: https://doi.org/10.1145/222124.222149.

[8] Shing Chi Cheung and Jeff Kramer. "Checking Safety Properties Using Compositional Reachability Analysis". In: *ACM Trans. Softw. Eng. Methodol.* 8.1 (Jan. 1999), pp. 49–78. ISSN: 1049-331X. DOI: 10.1145/295558.295570. URL: https://doi.org/10.1145/295558.295570.

[9] Shing Chi Cheung and Jeff Kramer. "Context constraints for compositional reachability analysis". In: *ACM Trans. Softw. Eng. Methodol.* 5.4 (Oct. 1996), pp. 334–377. ISSN: 1049-331X. DOI: 10.1145/235321.235323. URL: https://doi.org/10.1145/235321.235323.

[10] E.M. Clarke et al. *Model Checking, second edition*. Cyber Physical Systems Series. MIT Press, 2018. ISBN: 9780262349451. URL: https://books.google.com/books?id=qJl8DwAAQBAJ.

[11] Jamieson M. Cobleigh, George S. Avrunin, and Lori 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*. ISSTA '06. Portland, Maine, USA: Association for Computing Machinery, 2006, pp. 97–108. ISBN: 1595932631. DOI: 10.1145/1146238.1146250. URL: https://doi.org/10.1145/1146238.1146250.

[12] Jamieson M. Cobleigh, Dimitra Giannakopoulou, and Corina S. PĂsĂreanu. "Learning Assumptions for Compositional Verification". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Hubert Garavel and John Hatcliff. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 331–346. ISBN: 978-3-540-36577-8.

[13] Denis Cousineau et al. "TLA+ Proofs". In: *Proceedings of the 18th International Symposium on Formal Methods (FM 2012), Dimitra Giannakopoulou and Dominique Mery, editors. Springer-Verlag Lecture Notes in Computer Science* 7436 (Jan. 2012), pp. 147–154. URL: https://www.microsoft.com/en-us/research/publication/tla-proofs/.

[14] D. Giannakopoulou, C.S. Pasareanu, and H. Barringer. "Assumption generation for software component verification". In: *Proceedings 17th IEEE International Conference on Automated Software Engineering,* 2002, pp. 3–12. DOI: 10.1109/ASE.2002.1114984.

[15] Dimitra Giannakopoulou, Kedar S. Namjoshi, and Corina S. Păsăreanu. "Compositional Reasoning". In: *Handbook of Model Checking*. Ed. by Edmund M. Clarke et al. Cham: Springer International Publishing, 2018, pp. 345–383. ISBN: 978-3-319-10575-8. DOI: 10.1007/978-3-319-10575-8_12. URL: https://doi.org/10.1007/978-3-319-10575-8_12.

[16] Susanne Graf and Bernhard Steffen. "Compositional Minimization of Finite State Systems". In: *Proceedings of the 2nd International Workshop on Computer Aided Verification*. CAV '90. Berlin, Heidelberg: Springer-Verlag, 1990, pp. 186–196. ISBN: 3540544771.

[17] Jim Gray and Leslie Lamport. "Consensus on Transaction Commit". In: *ACM Trans. Database Syst.* 31.1 (Mar. 2006), pp. 133–160. ISSN: 0362-5915. DOI: 10.1145/1132863.1132867. URL: https://doi.org/10.1145/1132863.1132867.

[18] Anubhav Gupta, Kenneth L. McMillan, and Zhaohui Fu. "Automated Assumption Generation for Compositional Verification". In: *Proceedings of the 19th International Conference on Computer Aided Verification*. CAV'07. Berlin, Germany: Springer-Verlag, 2007, pp. 420–432. ISBN: 9783540733676.

[19] C. A. R. Hoare. "Communicating Sequential Processes". In: *Commun. ACM* 21.8 (Aug. 1978), pp. 666–677. ISSN: 0001-0782. DOI: 10.1145/359576.359585. URL: https://doi.org/10.1145/359576.359585.

[20] Cliff Jones. "Specification and Design of (Parallel) Programs." In: vol. 83. Jan. 1983, pp. 321–332.

[21] Igor Konnov, Jure Kukovec, and Thanh-Hai Tran. "TLA+ Model Checking Made Symbolic". In: *Proc. ACM Program. Lang.* 3.OOPSLA (Oct. 2019). DOI: 10.1145/3360549. URL: https://doi.org/10.1145/3360549.

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

[23] Jawahar Malhotra et al. "Winston: A Tool for Hierarchical Design and Simulation of Concurrent Systems". In: *Specification and Verification of Concurrent Systems*. Ed. by C. Rattray. London: Springer London, 1990, pp. 140–152. ISBN: 978-1-4471-3534-0.

[24] Björn Metzler, Heike Wehrheim, and Daniel Wonisch. "Decomposition for Compositional Verification". In: *Formal Methods and Software Engineering*. Ed. by Shaoying Liu, Tom Maibaum, and Keijiro Araki. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 105–125. ISBN: 978-3-540-88194-0.

[25] R. Milner. *Communication and Concurrency*. Ph/AMA Series in Marketing. Prentice Hall, 1989. ISBN: 9780131149847. URL: https://books.google.com/books?id=S5UZAQAAIAAJ.

[26] Wonhong Nam, P. Madhusudan, and Rajeev Alur. "Automatic Symbolic Compositional Verification by Learning Assumptions". In: *Form. Methods Syst. Des.* 32.3 (June 2008), pp. 207–234. ISSN: 0925-9856. DOI: 10.1007/s10703-008-0055-8. URL: https://doi.org/10.1007/s10703-008-0055-8.

[27] Rodrigo Otoni et al. "Symbolic Model Checking for TLA+ Made Faster". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Sriram Sankaranarayanan and Natasha Sharygina. Cham: Springer Nature Switzerland, 2023, pp. 126–144. ISBN: 978-3-031-30823-9.

[28] Corina Păsăreanu et al. "Learning to divide and conquer: Applying the L*algorithm to automate assume-guarantee reasoning". In: *Formal Methods in System Design* 32 (June 2008), pp. 175–205. DOI: 10.1007/s10703-008-0049-6.

[29] Corina S. Păsăreanu and Dimitra Giannakopoulou. "Towards a Compositional SPIN". In: *Model Checking Software*. Ed. by Antti Valmari. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 234–251. ISBN: 978-3-540-33103-2.

[30] Amir Pnueli. "In Transition From Global to Modular Temporal Reasoning about Programs". In: *Logics and Models of Concurrent Systems*. Ed. by Krzysztof R. Apt. Berlin, Heidelberg: Springer Berlin Heidelberg, 1985, pp. 123–144. ISBN: 978-3-642-82453-1.

[31] *Recomp-Verify Research Prototype Model Checker for TLA+*. https://github.com/cmu-soda/recomp-verify. Accessed: 2024-05-12.

[32] K.K. Sabnani, A.M. Lapone, and M.U. Uyar. "An algorithmic procedure for checking safety properties of protocols". In: *IEEE Transactions on Communications* 37.9 (1989), pp. 940–948. DOI: 10.1109/26.35374.

[33] William Schultz, Ian Dardik, and Stavros 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.

[34] K.-C. Tai and P.V. Koppol. "Hierarchy-based incremental analysis of communication protocols". In: *1993 International Conference on Network Protocols*. 1993, pp. 318–325. DOI: 10.1109/ICNP.1993.340896.

[35] Mark Weiser. "Programmers Use Slices When Debugging". In: *Commun. ACM* 25.7 (July 1982), pp. 446–452. ISSN: 0001-0782. DOI: 10.1145/358557.358577. URL: https://doi.org/10.1145/358557.358577.

[36] Wei Jen Yeh and Michal Young. "Compositional reachability analysis using process algebra". In: *Proceedings of the Symposium on Testing, Analysis, and Verification*. TAV4. Victoria, British Columbia, Canada: Association for Computing Machinery, 1991, pp. 49–59. ISBN: 089791449X. DOI: 10.1145/120807.120812. URL: https://doi.org/10.1145/120807.120812.

[37] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. "Model Checking TLA+ Specifications". In: *Correct Hardware Design and Verification Methods*. Ed. by Laurence Pierre and Thomas Kropf. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 54–66. ISBN: 978-3-540-48153-9.

[38] Hao Zheng. "Compositional Reachability Analysis for Efficient Modular Verification of Asynchronous Designs". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 29.3 (2010), pp. 329–340. DOI: 10.1109/TCAD.2009.2035544.

[39] Hao Zheng. "Local state space construction for compositional verification of concurrent systems". In: *Proceedings of the 2014 International SPIN Symposium on Model Checking of Software*. SPIN 2014. San Jose, CA, USA: Association for Computing Machinery, 2014, pp. 11–19. ISBN: 9781450324526. DOI: 10.1145/2632362.2632366. URL: https://doi.org/10.1145/2632362.2632366.

[40] Hao Zheng et al. "Compositional Model Checking of Concurrent Systems". In: *IEEE Transactions on Computers* 64.6 (2015), pp. 1607–1621. DOI: 10.1109/TC.2014.2329701.

Appendix A
Details of Parallel Composition in TLA+

We define // in the usual way, by synchronizing common actions between specifications and interleaving all others actions.

```
──── MODULE T1 ────
VARIABLES
  msgs, tmState, tmPrepared
Init ≜
  ∧ msgs = {}
  ∧ tmState = "init"
  ∧ tmPrepared = {}
SndPrepare(rm) ≜
  ∧ msgs' = msgs ∪
      {[type ↦ "Prepared",
        theRM ↦ rm]}
  ∧ UNCHANGED
      ⟨tmState, tmPrepared⟩
RcvPrepare(rm) ≜
  ∧ [type ↦ "Prepared",
      theRM ↦ rm] ∈ msgs
  ∧ tmState = "init"
  ∧ tmPrepared' =
      tmPrepared ∪ {rm}
  ∧ UNCHANGED
      ⟨msgs, tmState⟩
───────────────────
```

```
──── MODULE T2 ────
VARIABLES
  tmState, tmPrepared
Init ≜
  ∧ tmState = "init"
  ∧ tmPrepared = {}
RcvPrepare(rm) ≜
  ∧ tmState = "init"
  ∧ tmPrepared' =
      tmPrepared ∪ {rm}
  ∧ UNCHANGED ⟨tmState⟩
───────────────────
```

Fig. 6: Intermediate decomposed specifications T_1 and T_2 in the TP example.

Definition 2 (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. 2 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 covered in [22].

The following theorem shows that parallel composition behaves exactly as we expect if we convert a TLA⁺ specification to an LTS.

Theorem 2. $[\![\text{LTS}(S /\!/ T)]\!]_\alpha = [\![\text{LTS}(S) \parallel \text{LTS}(T)]\!]_\alpha$.

Before proving this theorem, we first introduce more general semantics for TLA⁺ specifications, namely action-state-based semantics.

We define an action-state-based behavior σ as an infinite sequence of action-state pairs: $(\epsilon, b_0)(a_1, b_1)(a_2, b_2)\ldots$, where ϵ is the empty action. We use the notation σ_i to access the ith pair of σ. Notice that we require the initial action to be empty, hence $\sigma_0 = (\epsilon, b_0)$. Let ϕ be an arbitrary TLA⁺ formula, let ψ 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 σ 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 αS 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 1. $[\![S /\!/ T]\!]_{\alpha\beta} = [\![S]\!]_{\alpha\beta} \cap [\![T]\!]_{\alpha\beta}$

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):

$$\begin{aligned} \forall i \in \mathbb{N} \;:\; &\vee \big(a_{i+1} \in \alpha S \wedge b_{i+1}' \models (b_i \wedge S!a_{i+1})\big) \\ &\vee \big(a_{i+1} \notin \alpha S \wedge b_{i+1}' \models (b_i \wedge S!vars' = S!vars)\big) \end{aligned} \tag{1}$$

We will prove equation (1)–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 (1) 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 (1) 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

$(S!\mathit{vars}' = S!\mathit{vars}) \wedge (T!\mathit{vars}' = T!\mathit{vars})$. Thus we see that $b'_{i+1} \models b_i \wedge (S!\mathit{vars}' = S!\mathit{vars}) \wedge (T!\mathit{vars}' = T!\mathit{vars})$ by the action-state-based semantics of TLA$^+$. This implies equation (1) 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 $\sigma \vdash (S /\!/ T)!\mathit{Spec}$. Recall that Spec has the form $\mathit{Init} \wedge \Box[\mathit{Next}]_{\mathit{vars}}$. Hence, we break the proof obligation into two tasks, one for each conjunct. The first task is to show $\sigma \vdash (S /\!/ T)!\mathit{Init}$, or equivalently $\sigma \vdash S!\mathit{Init} \wedge T!\mathit{Init}$. Because $\sigma \vdash S!\mathit{Spec}$ and $\sigma \vdash T!\mathit{Spec}$, we see that $\sigma \vdash S!\mathit{Init}$ and $\sigma \vdash T!\mathit{Init}$; thus the first task is proved. The second task is to show that $\sigma \vdash \Box[(S /\!/ T)!\mathit{Next}]_{\mathit{vars}}$. By the definition of the action-state-based semantics for TLA$^+$, this is equivalent to showing:

$$\begin{aligned} &\forall i \in \mathbb{N} \ : \\ &\vee \big(a_{i+1} \in \alpha(S /\!/ T) \wedge b'_{i+1} \models (b_i \wedge (S /\!/ T)!a_{i+1})\big) \\ &\vee \big(a_{i+1} \notin \alpha(S /\!/ T) \wedge b'_{i+1} \models \\ &\qquad (b_i \wedge (S /\!/ T)!\mathit{vars}' = (S /\!/ T)!\mathit{vars})\big) \end{aligned} \tag{2}$$

We will prove equation (2) 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 (2) 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 $S!a_{i+1} \wedge (T!\mathit{vars}' = T!\mathit{vars})$. Furthermore, $b'_{i+1} \models b_i \wedge S!a_{i+1}$ and $b'_{i+1} \models b_i \wedge (T!\mathit{vars}' = T!\mathit{vars})$. Therefore we conclude $b'_{i+1} \models b_i \wedge S!a_{i+1} \wedge (T!\mathit{vars}' = T!\mathit{vars})$ which implies equation (2) 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 $(S!\mathit{vars}' = S!\mathit{vars}) \wedge (T!\mathit{vars}' = T!\mathit{vars})$. Furthermore, $b'_{i+1} \models b_i \wedge (S!\mathit{vars}' = S!\mathit{vars})$ and $b'_{i+1} \models b_i \wedge (T!\mathit{vars}' = T!\mathit{vars})$. Therefore we conclude $b'_{i+1} \models b_i \wedge ((S!\mathit{vars} \circ T!\mathit{vars})' = (S!\mathit{vars} \circ T!\mathit{vars}))$. Since $(S /\!/ T)!\mathit{vars} = (S!\mathit{vars} \circ T!\mathit{vars})$, this implies equation (2) for this case.

∎

Lem. 1 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. 2.

Lemma 2. *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, σ_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. 2.

Theorem 2. $[\![\mathrm{LTS}(S /\!/ T)]\!]_\alpha = [\![\mathrm{LTS}(S) \parallel \mathrm{LTS}(T)]\!]_\alpha$.

Proof.

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

Where the second equality is due to Lem. 1, the third equality holds because S and T do not share state variables, and the fifth equality is due to Lem. 2. ∎

Appendix B
Static Specification Reduction

In Sec. IV, we point out that it is possible to statically detect a class of components that are unnecessary for verification. The class of unnecessary components is those whose alphabets cannot affect the actions of C_1 in any way, and hence cannot prevent the system from reaching an error. More formally, we first define $\hat{\alpha} Z$, where Z is a set of specifications, to be the union of each specification's alphabet, i.e. $\hat{\alpha} Z = \bigcup \{\hat{\alpha} C \mid C \in Z\}$. Then the set of unnecessary components is $\{C_1, \ldots, C_n\} - \bigcup_i X_i$, where $X_0 = \{C_1\}$ and $X_{i+1} = \{C_j \mid \hat{\alpha} C_j \cap \hat{\alpha} X_i \neq \emptyset\}$. We show this formally in Thm. 3.

Before presenting Thm. 3, we define $states(\sigma)$ on an action-state-based behavior σ 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: $states(B) = \{states(\sigma) \mid \sigma \in B\}$.

Theorem 3. *Let* $X_0 = \{C_1\}$, $X_{i+1} = \{C_j \mid \hat{\alpha} C_j \cap \hat{\alpha} X_i \neq \emptyset\}$, *and* $R = \{C_1, \ldots, C_n\} - \bigcup_i X_i$. *We refer to the components in* R *as* $R_1, \ldots, R_\rho$. *Then* $S \models P$ *if and only if* $R_1 /\!/ \cdots /\!/ R_\rho \models P$.

Proof. Let $U = \bigcup_i X_i$, and we will refer to the components in U as $U_1, \ldots, U_\upsilon$. Let $R_c = R_1 /\!/ \cdots /\!/ R_\rho$ and $U_c = U_1 /\!/ \cdots /\!/ U_\upsilon$. We note that $\hat{\alpha} R_c \cap \hat{\alpha} U_c = \emptyset$, which follows from the definition of X_{i+1}. Also, because R_c is composed of C_1, $\beta P \subseteq \beta R_c$ by property (P2) of Thm. 4. Because $\beta P \subseteq \beta R_c$ and components do not share state variables, we can therefore conclude that $\beta P \cap \beta U_c = \emptyset$. Now we have:

$$\begin{aligned} S \models P &\iff R_c /\!/ U_c \models P \\ &\iff states(\llbracket R_c /\!/ U_c \rrbracket_{\alpha\beta}) \subseteq \llbracket P \rrbracket_\beta \\ &\iff states(\llbracket R_c \rrbracket_{\alpha\beta} \cap \llbracket U_c \rrbracket_{\alpha\beta}) \subseteq \llbracket P \rrbracket_\beta \\ &\iff states(\llbracket R_c \rrbracket_{\alpha\beta}) \subseteq \llbracket P \rrbracket_\beta \\ &\iff R_c \models P \end{aligned}$$

In the equation above, the third biconditional is due to Lem. 1. The fourth biconditional follows for two reasons. The first reason is that $\beta P \cap \beta U_c = \emptyset$, and therefore the states in each behavior of $\llbracket U_c \rrbracket_{\alpha\beta}$ will not restrict the values of the variables in βP. The second reason is that $\hat{\alpha} R_c \cap \hat{\alpha} U_c = \emptyset$; this implies $\alpha R_c \cap \alpha U_c = \emptyset$, so the actions in each behavior of $\llbracket U_c \rrbracket_{\alpha\beta}$ will not restrict any behavior in $\llbracket R_c \rrbracket_{\alpha\beta}$. ☐

Example 5. We now introduce an extension to TP, $TPCounter$. This specification is identical to TP, except that it includes one more state variable $counter$ and one more action $Increment$. In the initial state of $TPCounter$, $counter$ 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.

Consider model checking $TPCounter \models Consistent$ with Alg. 1. The symbolic decomposition stage produces five components for $TPCounter$: RM, Env, TM_1, TM_2, and $Counter$, where $Counter$ has only the state variable $counter$ and the action $Increment$. $Counter$ is the only specification with the action $Increment$ and, therefore, does not synchronize with any actions in the other specifications. Hence, $Counter$ cannot affect the safety of C_1 in any way, and is unnecessary for verification.

More formally, $X_0 = \{RM\}$, $X_1 = \{RM, Env\}$, $X_2 = \{RM, Env, TM_1, TM_2\}$, $X_3 = \{RM, Env, TM_1, TM_2\}$, etc. Therefore, $\bigcup_i X_i = \{RM, Env, TM_1, TM_2\}$ and $\{RM, Env, TM_1, TM_2, Counter\} - \bigcup_i X_i = \{Counter\}$. We may therefore omit $Counter$ from the domain of f when solving the constraint satisfaction problem in the symbolic composition stage.

Appendix C
Decomposition Correctness and Slicing Procedure

A. Decomposition Correctness

The following theorem shows that the decomposition algorithm adheres to properties (P1) and (P2).

Theorem 4. *Algorithm 2 ensures (P1)* $S = C_1 /\!/ \cdots /\!/ C_n$ *and (P2)* $\beta C_1 \subseteq \beta P$ *upon termination.*

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 βP. Second, $\beta P \subseteq \beta C_1$ because $C_1 = \text{SLICE}(S, V_C)$ will contain exactly the state variables in V_C. ☐

B. 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 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
Compositional Verification Correctness with CRA

We now present the following theorem which shows the correctness of verification using a CRA style algorithm with short-circuiting.

Theorem 5. $\text{LTS}(D_1) \parallel \ldots \parallel \text{LTS}(D_m) \models \mathcal{P}$ *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 π state and

procedure $\text{SLICE}(S, V)$
  $T!vars \leftarrow \epsilon\big(S!vars \cap V\big)$
  $T!Init \leftarrow \text{SLICE-REC}(S!Init, V)$
  $acts \leftarrow \{\text{SLICE-REC}(A, V) \mid A \in \hat{\alpha}S\}$
  $T!Next \leftarrow \epsilon\big(\bigvee\{A \in acts \mid A \neq \text{DEL}\}\big)$
  return T
procedure $\text{SLICE-REC}(e, V)$
  match e **with**
  $\epsilon\big(op(p) \triangleq e_1\big) \rightarrow$
    $E \leftarrow \text{SLICE-REC}(e_1, V)$
    if $E = DEL$ **then**
      return DEL
    else
      return $\epsilon\big(op(p) \triangleq E\big)$

  $\epsilon\big(\exists x \in d \ : \ e_1\big) \rightarrow$
    $E \leftarrow \text{SLICE-REC}(e_1, V)$
    if $E = \text{DEL}$ **then**
      return DEL
    else
      return $\epsilon\big(\exists x \in d \ : \ E\big)$
  $\epsilon\left(\bigwedge_{i \in I} c_i\right) \rightarrow$
    $C \leftarrow \{c_j \mid \beta c_j \subseteq V\}$
    if $C = \emptyset$ **then**
      return DEL
    else
      return $\epsilon\left(\bigwedge C_j\right)$

Fig. 7: Slicing procedure definitions.

therefore neither will $Reach(\text{LTS}(D_P, P) \parallel \text{LTS}(D_1) \parallel \ldots \parallel \text{LTS}(D_m))$. $\square$

APPENDIX E
SOUNDNESS PROOF FOR ALG. 1

Theorem 1. *$S \models P$ if and only if $\text{LTS}(D_1) \parallel \ldots \parallel \text{LTS}(D_m) \models \mathcal{P}$.*

Proof. We show the majority of the proof in Fig. 8, in which we include the reason for each biconditional on the right hand side. The third biconditional in the figure holds when f is total. For the case when f is partial, the biconditional holds by Thm. 3 in Appendix B. Finally, the theorem follows because $\mathcal{P} = \text{PROP}(D_P, P)$. $\square$

APPENDIX F
DETAILS FOR THE DATA FLOW PARTIAL ORDER HEURISTIC

We now define the data flow partial order as follows. First, recall in Appendix B that we defined $X_0 = \{C_1\}$ and $X_{i+1} = \{C_j \mid \hat{\alpha}C_j \cap \hat{\alpha}X_i \neq \emptyset\}$. The X_i's cumulatively build the set of all specifications that interact with each other through shared actions. We now capture the sets that contain only the *new* specifications added to each X_{i+1}: $E_0 = X_0$ and $E_{i+1} = X_{i+1} \setminus X_i$.

Next, we are interested in the *data flow* from each specification in E_i to specifications in E_{i+1}; intuitively, this shows how how far removed each specification C_i is from impacting the variables in C_1, and hence impacting verification (by property (P2) of decomposition). We capture this data flow by letting $F_0 = \emptyset$ and:

$$F_{i+1} = \{(C_j, C_k) \mid C_j \in E_i \text{ and } C_k \in E_{i+1} \text{ and } \hat{\alpha}C_j \cap \hat{\alpha}C_k \neq \emptyset\}$$

Finally, let $F = \bigcup_i F_i$, then we define the data flow partial order $\preccurlyeq$ to be the reflexive transitive closure of F.

Example 6. Consider F from the construction of the data flow partial order above. For TP and $Consistent$, $F = \{(RM, Env), (Env, TM_1), (Env, TM_2)\}$, and hence the data flow partial order for TP is the reflexive transitive closure of F. Intuitively, this partial order shows that TM_1 and TM_2 are "one step removed" from affecting the variables in RM (i.e. the variables $\beta Consistent$ needed for verification) because TM_1 and TM_2 can only influence RM's variables by interacting with the Env component.

The state variables in TM_2 occur fewer times syntactically than the variables in TM_1 in TP (the entire specification), and hence the total ordering resulting from Sec. V-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 $f(RM) = f(Env) = f(TM_2) = f(TM_1) = d_P$.

We now show that the X_i's converge within a bound of n.

Lemma 3. $X_n = X_{n+1}$

Proof. A simple inductive argument shows that whenever $X_i \neq X_{i+1}$, $|X_{i+1}| \geq i + 1$. Now suppose, for the sake of contradiction, 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. $\square$

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 the static class of necessary components ($\bigcup_i X_i$) *and* provides an ordering over these components.

Theorem 6. *Let $G = \{C_i \mid \exists C_j : C_i \preccurlyeq C_j \text{ or } C_j \preccurlyeq C_i\}$ 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 specifications that we choose to keep during static specification reduction in Appendix B.*

$$
\begin{array}{rll}
S \models P & \Longleftrightarrow \pi \in \mathit{Reach}(\textsc{lts}(S, P)) & P \text{ is a safety property} \\
& \Longleftrightarrow \pi \in \mathit{Reach}(\textsc{lts}(C_1 /\!\!/ \cdots /\!\!/ C_n, P)) & \text{Thm. 4 property (P1)} \\
& \Longleftrightarrow \pi \in \mathit{Reach}(\textsc{lts}(D_P /\!\!/ D_1 /\!\!/ \cdots /\!\!/ D_m, P)) & \text{Definition for } D_j\text{s} \\
& \Longleftrightarrow \pi \in \mathit{Reach}(\textsc{lts}(D_P, P) \parallel \textsc{lts}(D_1, P) \parallel \ldots \parallel \textsc{lts}(D_m, P)) & \text{Thm. 2} \\
& \Longleftrightarrow \pi \in \mathit{Reach}(\textsc{lts}(D_P, P) \parallel \textsc{lts}(D_1) \parallel \ldots \parallel \textsc{lts}(D_m)) & \text{Thm. 4 property (P2)} \\
& & \text{and } f(C_1) = d_P \\
& \Longleftrightarrow \textsc{lts}(D_1) \parallel \ldots \parallel \textsc{lts}(D_m) \models \textsc{prop}(D_P, P) & P \text{ is a safety property}
\end{array}
$$

Fig. 8: Equation including explanations that are used for proving Thm. 1.

Proof. By Lem. 3, the X_i sequence converges by the nth term, and therefore:

$$\bigcup_i X_i = \bigcup_{1 \le i \le n} X_i$$

By the definition of each E_i, we have $\bigcup_i X_i = \bigcup_i E_i$, where the bounds on i in each union are $1 \le i \le n$.

Next, based on the definition for X_{i+1}, an alternate formulation for E_{i+1} is:

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

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 $\mathit{proj}_1(F_{i+1})$ for the projection onto the first element.

$$
\begin{aligned}
\mathit{proj}_1(F_{i+1}) = \{ C_j \mid{} & C_j \in E_i \text{ and} \\
& \exists C_k \in E_{i+1} : \hat{\alpha} C_j \cap \hat{\alpha} C_k \neq \emptyset \} \\
= E_i \cap{} & \{ C_j \mid \hat{\alpha} C_j \cap \hat{\alpha} E_{i+1} \neq \emptyset \} \\
= E_i &
\end{aligned}
$$

Where the final equality is due to the alternate definition of E_{i+1}. Therefore, $\bigcup_{i \ge 0} E_i = \bigcup_{i \ge 1} \mathit{proj}_1(F_i)$. We can further conclude $\bigcup_i E_i = \bigcup_i \mathit{proj}_1(F_i)$ because $F_0 = \emptyset$.

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 $\mathit{proj}_2(F_{i+1})$ for the projection onto the second element.

$$
\begin{aligned}
\mathit{proj}_2(F_{i+1}) &= \{ C_k \mid C_k \in E_{i+1} \text{ and} \\
&\qquad \exists C_j \in E_i : \hat{\alpha} C_j \cap \hat{\alpha} C_k \neq \emptyset \} \\
&= \{ C_k \mid C_k \in E_{i+1} \text{ and } \hat{\alpha} E_i \cap \hat{\alpha} C_k \neq \emptyset \} \\
&= E_{i+1} \cap \{ C_k \mid \hat{\alpha} E_i \cap \hat{\alpha} C_k \neq \emptyset \} \\
&= E_{i+1}
\end{aligned}
$$

Where the final equality is due to the alternate definition of E_{i+1}. Therefore, $\bigcup_{i \ge 1} E_i = \bigcup_{i \ge 1} \mathit{proj}_2(F_i)$.

Finally, we see:

$$\bigcup_i X_i = \bigcup_{1 \le i \le n} E_i = \bigcup_i \mathit{proj}_1(F_i) \cup \bigcup_i \mathit{proj}_2(F_i) = G$$

$\square$