

# 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 have implemented our techniques in a model checker for the TLA<sup>+</sup> language. In our experiments, we show that our tool achieves competitive performance with TLC—a well-known model checker for TLA<sup>+</sup>—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  $C_1, \dots, C_n$ . Traditionally, a compositional verification algorithm is applied to these components to verify a system level property  $P$ , as shown in Fig. 1a. However, verifying these components 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 [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, selecting components for efficient verification reduces to finding a suitable recomposition map. Therefore, we propose a technique for automatically selecting 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<sup>+</sup> language [22]. In order to bring compositional verification to TLA<sup>+</sup>, we additionally propose a novel parallel composition operator for the language. We evaluate our techniques by comparing Recom-Verify to TLC [23], a well-known model checker for TLA<sup>+</sup>. 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<sup>+</sup> specifications, and (4) a prototype model checker Recom-Verify that implements our algorithm, along with an evaluation of Recom-Verify against TLC on a benchmark of

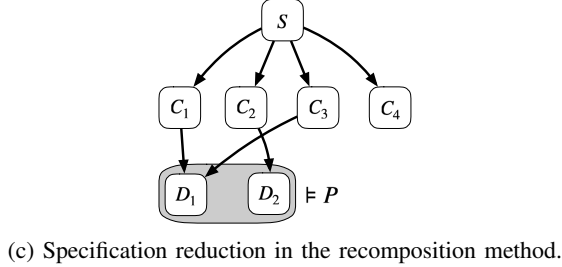
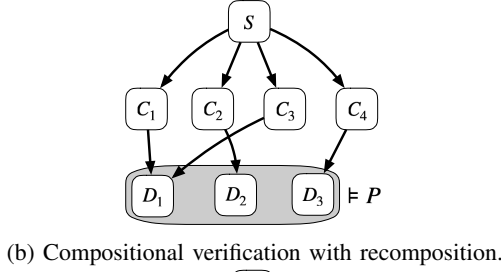
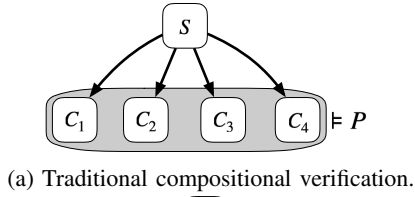


Fig. 1: Comparing traditional compositional verification against our recomposition method.

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<sup>+</sup> 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

## 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.

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<sup>+</sup>, 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<sup>+</sup> 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 a given finite instance of *TP* satisfies 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

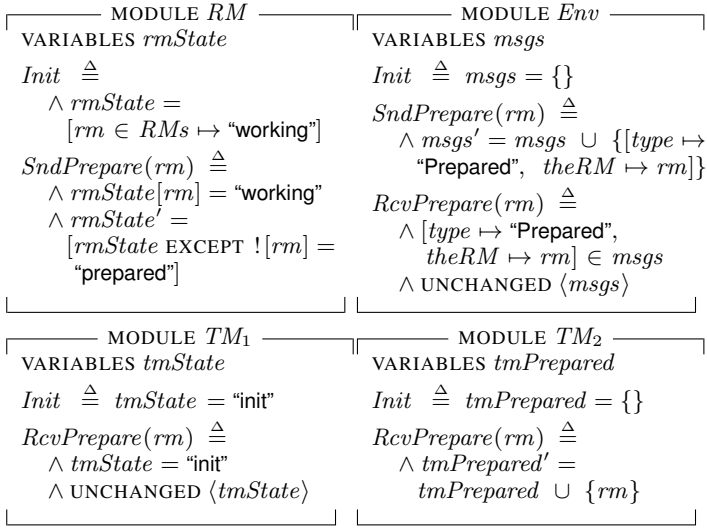


Fig. 3: A decomposition of *TP*. Standard operators such as *Spec*, *Next*, *vars*, etc. are omitted for brevity.

managers.

4) *Compositional Verification and Recomposition*: Consider the specifications *RM*, *Env*, *TM*<sub>1</sub>, and *TM*<sub>2</sub> 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.

Compositionally verifying the components above is more efficient than TLC, but *we can use recomposition to do even better*. Later, in example Ex. 2, we 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<sup>+</sup> 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<sup>+</sup> 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  $Reach(\mathcal{D})$

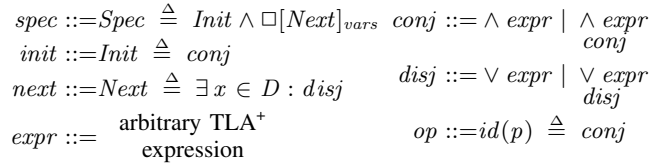


Fig. 4: Restricted TLA<sup>+</sup> grammar for this paper.

be the set of reachable states in  $\mathcal{D}$ . We define the parallel composition ( $\parallel$ ) over LTSs in the usual way by synchronizing on actions common to both alphabets and interleaving on all other actions [25].

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  $\llbracket \mathcal{D} \rrbracket_\alpha \subseteq \mathbb{A}^\omega$ . It is the case that  $\sigma \in \llbracket \mathcal{D} \rrbracket_\alpha$  if and only if there exists a sequence of states  $q_0, q_1, \dots \in Q^\omega$  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 steps two and three for assumption generation (Sec. 3) in [26]. 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  $\llbracket \mathcal{D} \rrbracket_\alpha \subseteq \llbracket \mathcal{P} \rrbracket_\alpha$ . Note that our LTS semantics (with stuttering) properly handles alphabet refinement, and therefore it is unnecessary to consider alphabet restriction [25] in our definition of property satisfaction.

#### B. TLA<sup>+</sup>

In this paper we will refer to a TLA<sup>+</sup> specification *S* as a syntactic entity that consists of constants, variables 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,  $\square$  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<sup>+</sup> 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  $RM = \{\text{"rm1"}\}$ , then  $\hat{\alpha}TP_1 = \hat{\alpha}TP = \{SndPrepare, RcvPrepare\}$  and  $\alpha TP_1 = \{SndPrepare(\text{"rm1"}), RcvPrepare(\text{"rm1"})\}$ . 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<sup>+</sup> formula, we first define a state as an assignment to all state variables. Then, the semantics of a TLA<sup>+</sup> formula is a set of behaviors, where a behavior is an infinite sequence of states. We indicate state-based semantics of a TLA<sup>+</sup> formula  $F$  as  $\llbracket F \rrbracket_\beta$ , the set of behaviors that satisfy  $F$ . For a TLA<sup>+</sup> specification  $S$ , we will often abbreviate  $\llbracket S.Spec \rrbracket_\beta$  to simply  $\llbracket S \rrbracket_\beta$ . Given a TLA<sup>+</sup> property  $P$ , we say  $S$  satisfies  $P$  ( $S \models P$ ) exactly when  $\llbracket S \rrbracket_\beta \subseteq \llbracket P \rrbracket_\beta$ .

We define the operator  $LTS(S)$ , which converts a TLA<sup>+</sup> specification  $S$  into an LTS  $\mathcal{D}$ .  $LTS(S)$  can be realized by generating the full state graph for  $S$  and then labeling its edges with the concrete actions  $\alpha S$  such that  $\alpha \mathcal{D} = \alpha S$ . Additionally, we define two operators for converting TLA<sup>+</sup> properties to an LTS. The first operator,  $ERR(S, P)$ , constructs an error LTS for  $S$  where violations of  $P$  lead to a  $\pi$  state. The second operator,  $PROP(S, P)$ , builds a property LTS for  $S$  where no violation of  $P$  is possible.  $PROP(S, P)$  can be constructed from  $ERR(S, P)$ , as pointed out in Sec. III-A.

### C. CRA-Style Compositional Verification

In this paper, we consider a style of compositional verification called *compositional reachability analysis* (CRA) [4], [8], [27]. 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<sup>+</sup>

In this section, we introduce a new parallel composition operator over TLA<sup>+</sup> specifications. The operator is central to our recomposition algorithm and will allow us to define

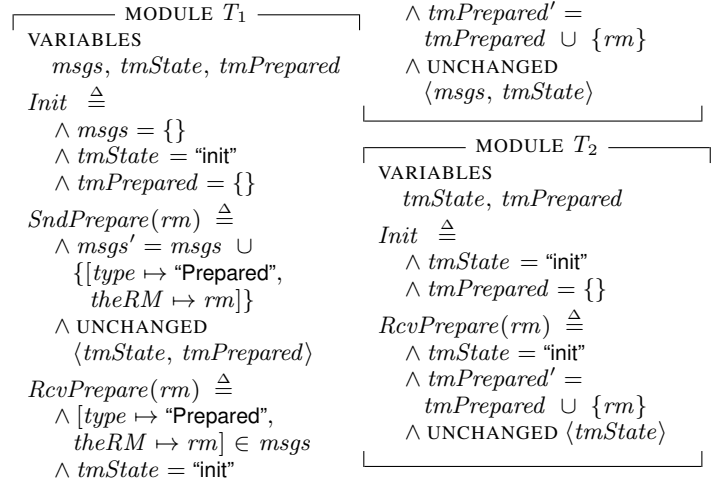


Fig. 5: Intermediate decomposed specifications  $T_1$  and  $T_2$  in the  $TP$  example.

concepts such as decomposition and recomposition in Sec. V. The new operator is syntactic; in other words, the definition is entirely in terms of TLA<sup>+</sup> syntax, and does not involve explicitly enumerating the state space. To avoid confusion between the parallel composition operator  $\parallel$  over LTSs (Sec. III-A), we will denote the TLA<sup>+</sup> parallel composition operator using  $\parallel$ . We will use the notation  $\parallel Z$  to denote the composition over a set of specifications  $Z$ . We now define  $\parallel$  in the usual way, by synchronizing common actions between specifications and interleaving all others actions [25].

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

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

Notice that Def. 1 defines parallel composition in terms of TLA<sup>+</sup> 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,  $TP = RM \parallel Env \parallel TM_1 \parallel TM_2$ . Furthermore, consider specifications  $T_1$  and  $T_2$  from Fig. 5. Notice that  $TP = RM \parallel T_1$ ,  $T_1 = Env \parallel T_2$ , and  $T_2 = TM_1 \parallel TM_2$ .

The following theorem shows that parallel composition behaves exactly as we expect if we convert a TLA<sup>+</sup> speci-

fication to an LTS. We prove this theorem using more general semantics for TLA<sup>+</sup> specifications, namely action-state-based semantics. We include a proof in Appendix A of our technical report [29].

**Theorem 1.**  $\llbracket \text{LTS}(S // T) \rrbracket_\alpha = \llbracket \text{LTS}(S) \parallel \text{LTS}(T) \rrbracket_\alpha$ .

## 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. Subsequently, we provide details for decomposition (Sec. V-B), static specification reduction (Sec. V-C), and compositional verification (Sec. V-D). Finally, we conclude this section with a correctness analysis of the algorithm in Sec. V-E.

### A. The Recompose-Verify Algorithm

1) *Algorithm Overview:* Our algorithm solves a model checking problem  $S \models P$ , where  $S$  and  $P$  are both written in TLA<sup>+</sup>. The algorithm begins by decomposing  $S$  into  $n$  components  $C_1, \dots, C_n$ , each of which is also a TLA<sup>+</sup> specification. The decomposition algorithm ensures two key properties upon termination: (P1)  $S = C_1 // \dots // C_n$  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  $D_1, \dots, D_m$ . These new components define the following compositional verification problem that is equivalent to the original:  $\text{LTS}(D_1) \parallel \dots \parallel \text{LTS}(D_m) \models \mathcal{P}$ , 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 (at least)  $C_1$ , 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  $f : \{C_1, \dots, C_n\} \rightarrow \{d_P, d_1, \dots, d_m\}$  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 total recomposition map and  $m$  is the number of  $D_j$  components. In other words, the recomposition strategy determines which recomposition map is used. In the remainder

of this section we assume  $\rho$  is given; we discuss recomposition strategy selection in Sec. VI.

---

### Algorithm 1 RECOMP-VERIFY

---

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

**Output:** If  $S \models P$

```

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

```

---

Alg. 1 begins by decomposing  $S$  into components on line 1. The strategy  $\rho$  selects a recomposition map on line 2, which is possibly statically reduced on line 3. We provide more detail for decomposition and static specification reduction in Sec. V-B and Sec. V-C respectively. Next, on lines 4-6, we perform recomposition using the recomposition map  $f$ . On line 4, 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 6, 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 7, we solve the compositional verification problem for the recomposed components ( $D_j$ 's); we provide more detail for this step in Sec. V-D.

**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 produces the components  $RM$ ,  $Env$ ,  $TM_1$ , and  $TM_2$  from Fig. 3. On line 2,  $\rho_{opt}$  chooses  $m = 2$  and  $f$  such that  $f(RM) = d_P$ ,  $f(Env) = f(TM_1) = d_1$ , and  $f(TM_2) = d_2$ . Static specification reduction on line 3 has no effect on  $f$  and  $m$ . Recomposition (lines 4-6) reduces the original model checking problem to  $\text{LTS}(Env // TM_1) \parallel \text{LTS}(TM_2) \models \text{PROP}(RM, \text{Consistent})$ , which we solve on line 7. 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 improves the model checking time from 1 minute 32 seconds to 51 seconds.

### 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)  $S = C_1 // \dots // C_n$  and (P2)  $\beta P \subseteq \beta C_1$ . We provide a correctness argument for these two properties in Sec. V-E.

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 3.** 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  $T_1$  (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.

---

**Algorithm 2** DECOMPOSE

---

**Input:** Specification  $S$ , Safety Property  $P$

**Output:**  $C_1, \dots, C_n$  with properties (P1) and (P2)

```

1:  $V_C, V_T \leftarrow \text{PARTITION}(S, \beta P)$ 
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:    $C_{i+1} \leftarrow \text{SLICE}(T_i, V_C)$ 
7:    $T_{i+1} \leftarrow \text{SLICE}(T_i, V_T)$ 
8:    $v \in \beta T_{i+1} \triangleright$  Nondeterministically choose a variable
9:    $V_C, V_T \leftarrow \text{PARTITION}(T_{i+1}, \{v\})$ 
10:   $i \leftarrow i + 1$ 
11:  $n \leftarrow i + 1, C_n \leftarrow T_i$ 
12: return  $C_1, \dots, C_n$ 
13: procedure  $\text{PARTITION}(T, V)$ 
14:    $V_C \leftarrow \text{FIX}(\text{OCCURS}_T, V)$ 
15:    $V_T \leftarrow \beta S - V_C$ 
16:   return  $V_C, V_T$ 
17: procedure  $\text{OCCURS}_S(V)$ 
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  $\text{FIX}(op, X)$ 
20:    $Y \leftarrow X \cup op(X)$ 
21:   if  $X = Y$  then return  $X$ 
22:   return  $Y \cup \text{FIX}(op, Y)$ 

```

---

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  $V_C = \text{FIX}(\text{OCCURS}_S, V)$  (line 14), where  $\text{FIX}$  invokes the  $\text{OCCURS}_S$  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 4.** Notice that  $\beta Consistent = \{rmState\}$  and  $\text{FIX}(\text{OCCURS}_{TP}, \{rmState\}) = \{rmState\}$ . 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  $var_2$ :

$$\begin{aligned} Action &\triangleq \wedge var_1' = \text{"val1"} \\ &\quad \wedge var_2' = \text{"val2"} \end{aligned}$$

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  $M_1 = \text{SLICE}(M, \{var_1\})$ ,  $M_2 = \text{SLICE}(M, \{var_2\})$ , and  $M = M_1 // M_2$ . In the  $TP$  example, this corresponds to  $TP = RM // T_1$  in Ex. 3. We include more details on slicing, including the definition for the slicing procedure, in Appendix B of our technical report [29].

### C. Static Specification Reduction

In Sec. V-A, we require recomposition strategies to produce a total recomposition map. Total recomposition maps apply verification to every component; however, in some cases, not every component is necessary for verification. Therefore, in the following paragraph, we introduce a technique for statically detecting a subset of components that are necessary for verification. In Alg. 1, the procedure  $\text{STATIC-REDUCE}(f, m)$  on line 3 restricts the domain of  $f$  to this subset and reduces the codomain and  $m$  accordingly so  $f$  remains surjective.

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 C of our technical report [29], we show that it is only necessary to consider the first  $n + 1$  terms—where  $n$  is the number of  $C_i$  components—when computing the union of the  $X_i$ 's. 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 5.** 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 original action 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<sub>1</sub>*, *TM<sub>2</sub>*, 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. STATIC-REDUCE will therefore omit *Counter* from the domain of any given recomposition map, causing Alg. 1 to successfully terminate.

#### D. 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.

#### E. Correctness Analysis

In this section, we show that Alg. 1 is sound but not complete. To establish this result, we first provide lemmas for

---

#### Algorithm 3 COMP-VERIFY

---

**Input:**  $D_1, \dots, D_m, D_P, P$

**Output:** If  $LTS(D_1) \parallel \dots \parallel LTS(D_m) \models \mathcal{P}$

```

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

```

---

the correctness of decomposition (Lem. 1), static specification reduction (Lem. 2), and compositional verification (Lem. 3). Next, we show that reducing the monolithic model checking problem to compositional verification is correct (Lem. 4). Finally, we present Thm. 2 that shows that Alg. 1 is sound.

**Lemma 1.** *Algorithm 2 ensures (P1)  $S = C_1 // \dots // C_n$  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:  $S = C_1 // \dots // C_i // T_i$ . 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  $C_1 = SLICE(S, V_C)$  will contain exactly the state variables in  $V_C$ .  $\square$

**Lemma 2.**  $S \models P$  if and only if  $(\bigcup_i X_i) \models P$ .

*Proof.* We prove this theorem in Appendix C of our technical report [29].  $\square$

**Lemma 3.**  $LTS(D_1) \parallel \dots \parallel LTS(D_m) \models \mathcal{P}$  if and only if there exists a  $k \in \{0 \dots m\}$  such that  $\pi \notin Reach(ERR(D_P, P) \parallel LTS(D_1) \parallel \dots \parallel LTS(D_k))$ .

*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  $\pi \notin Reach(ERR(D_P, P) \parallel LTS(D_1) \parallel \dots \parallel LTS(D_k))$ . Notice that, by construction, none of  $LTS(D_{k+1}), \dots, LTS(D_m)$  contains a  $\pi$  state, and therefore neither will  $Reach(ERR(D_P, P) \parallel LTS(D_1) \parallel \dots \parallel LTS(D_m))$ .  $\square$

**Lemma 4.**  $S \models P \iff LTS(D_1) \parallel \dots \parallel LTS(D_m) \models \mathcal{P}$ .

*Proof.*

$$S \models P \tag{1}$$

$$\iff \pi \in Reach(ERR(S, P)) \tag{2}$$

$$\iff \pi \in Reach(ERR(C_1 // \dots // C_n, P)) \tag{3}$$

$$\iff \pi \in Reach(ERR(D_P // D_1 // \dots // D_m, P)) \tag{4}$$

$$\iff \pi \in Reach(ERR(D_P, P) \parallel \dots \parallel ERR(D_m, P)) \tag{5}$$

$$\iff \pi \in Reach(ERR(D_P, P) \parallel \dots \parallel LTS(D_m)) \tag{6}$$

$$\iff LTS(D_1) \parallel \dots \parallel LTS(D_m) \models PROP(D_P, P) \tag{7}$$

Biconditional (2) holds because  $P$  is a safety property, (3) by Lem. 1 property (P1), (4) by the definition for each  $D_j$  (and Lem. 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  $\mathcal{P} = \text{PROP}(D_P, P)$ .  $\square$

**Theorem 2.** *If Alg 1 terminates, then it returns true (model checking succeeds) if and only if  $S \models P$ .*

*Proof.* The result follows from Lem. 3 and Lem. 4.  $\square$

While Thm. 2 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 a given component  $D_j$  may be infinite-state, in which case LTS construction 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 recombination strategies, i.e. choosing efficient recombination maps. Rather than finding a single recombination 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 recombination maps grows large as the number of components increases. We therefore propose a heuristic for pruning the search space of recombination maps in Sec. VI-A. We then choose a small portfolio of strategies based on this heuristic in Sec. VI-B.

### A. Recombination Map Reduction Heuristic

Any heuristic for pruning the search space of recombination 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 recombination maps that order  $D_j$  components that are least likely to be necessary for verification *last*.

Our heuristic is to choose recombination maps that respect the *data flow* partial order  $\preceq$  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 indexed sets  $X_i$  introduced in Sec. V-C. These sets cumulatively capture the components that may interact—either directly or indirectly—with the actions of  $C_1$ . First, we build new indexed sets  $E_i$  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 indexed sets  $F_i$  that capture the *data flow* from each component in  $E_i$  to the components in  $E_{i+1}$ . We define  $F_0 = \emptyset$  and:

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

Finally, let  $F = \bigcup_i F_i$ ; we define the data flow partial order  $\preceq$  to be the reflexive transitive closure of  $F$ . In Appendix D of our technical report [29], we show formally that the partial order refines the static specification reduction scheme from Sec. V-C.

**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  $\leq$ , i.e.  $\preceq \subseteq \leq$ . We build the total order by breaking ties between incomparable components  $C_i$  and  $C_j$  by requiring  $C_i \leq C_j$  if and only if  $C_i$ 's state variables have fewer syntactic appearances than  $C_j$ 's in the original specification  $S$ . In the case that the variables of  $C_i$  and  $C_j$  have the same number of appearances in  $S$ , we break the tie arbitrarily.

### B. Choosing a Portfolio of Strategies

In this section, we describe four recombination strategies that comprise our portfolio. For simplicity, we describe the strategies assuming that the components  $C_1, \dots, C_n$  have been reordered according to the total order  $\leq$  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.** In  $TP$ , the state variables of  $TM_2$  occur fewer times than the variables of  $TM_1$ . Therefore, the total ordering from Sec. VI-A is:  $RM, Env, TM_2, TM_1$ . Then, for each strategy, we have: (S1)  $m = 3$ ,  $f(RM) = d_P$ ,  $f(Env) = d_1$ ,  $f(TM_2) = d_2$ , and  $f(TM_1) = d_3$ ; (S2)  $m = 1$ ,  $f(RM) = d_P$ , and  $f(Env) = f(TM_2) = f(TM_1) = d_1$ ; (S3)  $m = 1$ ,  $f(RM) = f(Env) = f(TM_2) = d_P$ , and  $f(TM_1) = d_1$ ; and (S4)  $m = 0$  and  $f(RM) = f(Env) = f(TM_2) = f(TM_1) = d_P$ .

As a note regarding the correctness analysis from Sec. V-E, 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<sup>+</sup> 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 [30].

### B. Experiments

We evaluate *Recomp-Verify* against TLC on a benchmark of distributed protocols [31], plus the *tlc-twophase-counter* protocol that we introduce in Ex. 5. 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<sup>1</sup> and TLC<sup>4</sup> to respectively denote TLC run with one and four parallel threads; this is a built-in option for the tool. *Recomp-Verify*<sup>1</sup> is the version of our tool with one thread and hand-crafted maps, while *Recomp-Verify*<sup>4</sup> 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*<sup>4</sup> in the “Strat.” column in Fig. 6. Additionally, in the implementation of *Recomp-Verify*<sup>4</sup>, we use TLC<sup>1</sup> 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*<sup>1</sup> uses an optimal single-threaded strategy, yet is slower than TLC<sup>1</sup>—and therefore *Recomp-Verify*<sup>4</sup> 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*<sup>1</sup>, 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<sup>1</sup> enumerates at least as many states as *Recomp-Verify*<sup>1</sup> in every case. For six of the benchmarks that both tools verified, recomposition reduced the state size by *millions*

of states. Moreover, *Recomp-Verify*<sup>1</sup> short-circuits ( $k < m$ ) for eight benchmarks, each of which has a significantly smaller state space than TLC<sup>1</sup>. Finally, *Recomp-Verify*<sup>1</sup>—but not TLC<sup>1</sup>—was able to verify the one infinite state benchmark (*tlc-twophase-counter*) via static specification reduction.

In terms of verification speed, *Recomp-Verify*<sup>1</sup> and TLC<sup>1</sup> both outperform each other in fourteen benchmarks, and tie in five cases. However, *Recomp-Verify*<sup>1</sup> completes more benchmarks, verifying twenty-nine benchmarks while TLC<sup>1</sup> verifies twenty-four. Generally speaking, *Recomp-Verify*<sup>1</sup> is more performant on larger benchmarks; on benchmarks with over a million states, TLC<sup>1</sup> is faster in two cases while *Recomp-Verify*<sup>1</sup> 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<sup>4</sup> and *Recomp-Verify*<sup>4</sup> are similar to the single threaded versions. In terms of state space, *Recomp-Verify*<sup>4</sup> always enumerates the same number of (or fewer) states than TLC<sup>4</sup>, and also exhibits large savings in the millions for six benchmarks. We note that *Recomp-Verify*<sup>4</sup> short-circuited every time that *Recomp-Verify*<sup>1</sup> short-circuited, which showcases the effectiveness of the data flow heuristic.

In terms of verification speed, *Recomp-Verify*<sup>4</sup> is faster for eleven benchmarks, TLC<sup>4</sup> is faster for fourteen, and the tools tie for eight benchmarks. However, *Recomp-Verify*<sup>4</sup> verifies more benchmarks, completing thirty-two, while TLC<sup>4</sup> 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*<sup>4</sup> outperforms TLC<sup>4</sup> for large benchmarks and is competitive with TLC<sup>4</sup> for smaller ones, and therefore we answer RQ2 in the affirmative.

3) *Discussion*: In Fig. 6, the Strat. column shows that the winning strategy for *Recomp-Verify*<sup>4</sup> varies depending on the given benchmark. This observation suggests that using a portfolio of strategies may be necessary for efficient verification. Notably, among the benchmark problems, we found that the bottom heavy strategy (S2) did not perform well. Most likely, this is because the second component in this strategy is too large, and therefore misses opportunities for short-circuiting.

Ultimately, *Recomp-Verify* tends to be faster than TLC on benchmarks that have more opportunity for recomposition. We point out that *Recomp-Verify*<sup>1</sup> is faster than TLC<sup>1</sup> in *every* case where decomposition produced at least four components ( $n \geq 4$ ). The same is true for *Recomp-Verify*<sup>4</sup> and TLC<sup>4</sup> 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], [32], [12], [13], [14], [15], [16] and assume-guarantee reasoning [2], [3], [7], [17], [9], [33], [18], [19], [10], [11], [34], although other styles exist as well [35],

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

Fig. 6: Run time comparison between Recom-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.

[36]. Of these works, the ones most closely related to this paper automate decomposition for verification. Metzler et al. [18] and Cobleigh et al. [17] decompose systems into two components, after which they apply  $L^*$  style learning [37] 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 [38], 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 [39], [40] 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) [41] provides an alternative to model checking  $TLA^+$ . TLAPS proofs are manually constructed, but automatically verified by dispatching proof obligations to SMT solvers. Endive [31] 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 that different threads can share intermediate work to save time and memory.

In this paper, we focus on using recomposition for explicit-state model checking. However, recomposition may also be effective for other compositional verification tasks. For example, we plan to investigate whether recomposition can be used in combination with non-explicit verification techniques, e.g. using SMT solvers. 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](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. Wonsch, “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] 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>
- [26] 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.
- [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=S5UZAQAIAAJ>
- [29] I. Dardik, A. Porter, and E. Kang, “Recomposition: A new technique for efficient compositional verification,” 2024. [Online]. Available: <https://arxiv.org/abs/2408.03488>
- [30] “Recomp-verify research prototype model checker for tla+,” <https://github.com/cmua-soda/recomp-verify/tree/FMCAD24>, accessed: 2024-08-16.
- [31] 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.
- [32] 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.
- [33] C. Jones, “Specification and design of (parallel) programs.” vol. 83, 01 1983, pp. 321–332.
- [34] 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.
- [35] H. Barringer, D. Giannakopoulou, and C. Pasareanu, “Proof rules for automated compositional verification through learning,” 02 2003.
- [36] 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.
- [37] 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](https://doi.org/10.1016/0890-5401(87)90052-6)
- [38] 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>

- [39] 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>
- [40] 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.
- [41] 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/>