Local Proofs for Global Safety Properties

Ariel Cohen[1] and Kedar S. Namjoshi[2]

[1] New York University, arielc@cs.nyu.edu
[2] Bell Labs, kedar@research.bell-labs.com

Abstract. This paper explores locality in proofs of global safety properties of concurrent programs. Model checking on the full state space is often infeasible due to state explosion. A local proof, in contrast, is a collection of per-process invariants, which together imply the desired global safety property. Local proofs can be more compact than global proofs, but local reasoning is also inherently incomplete. In this paper, we present an algorithm for safety verification that combines local reasoning with gradual refinement. The algorithm gradually exposes facts about the internal state of components, until either a local proof or a real error is discovered. The refinement mechanism ensures completeness. Experiments show that local reasoning can have significantly better performance over the traditional reachability computation. Moreover, for some parameterized protocols, a local proof can be used as the basis of a correctness proof over *all* instances.

1 Introduction

The success achieved by model checking [6, 31] in various settings has always been tempered by the problem of state explosion [4]. Strategies based on abstraction and compositional analysis help to ameliorate the adverse effects of state explosion. This paper explores a particular combination of the two, which may be called "local reasoning". The context is the analysis of invariance properties of shared-variable, multi-process programs. Multi-threaded programs, and protocols for cache coherence and mutual exclusion can be modeled in this setting. This paper concentrates on invariance properties; more complex safety properties can be reduced to invariance checking by standard methods.

Model checking algorithms prove an invariance property through a reachability computation, computing an inductive assertion (the reachable states) that is defined over the full state vector. In contrast, a *local proof* of invariance for an asynchronous composition, $P_1||P_2||\ldots||P_n$, is given by a vector of assertions, $\{\theta_i\}$, one for each process, such that their conjunction is inductive, and implies the desired invariance property. Locality is enforced by syntactically limiting each assertion θ_i to the shared variables, X, and the local variables, L_i, of process P_i. The vector θ is called a *split invariant*.

In recent work [26], it is shown that a *strongest* split invariant exists, and can be computed as a least fixpoint. It is also shown that the split invariance formulation inherently encodes the principle of non-interference that is central to the compositional, deductive proof method of Owicki and Gries [27].

Local proofs can be more compact than global proofs, but local reasoning is also inherently incomplete: i.e., some valid properties do not have local proofs. This is because a split invariant over-approximates the set of reachable states, which may cause some unreachable error states to be included in the invariant. The over-approximation arises from the loose coupling between local process states: a joint constraint on L_i and L_j can be enforced only via X, through the term $\theta_i(X, L_i) \wedge \theta_j(X, L_j)$. Owicki and Gries showed that completeness can be recovered by adding auxiliary history variables to the shared state. Independently, Lamport showed in [23] that sharing all local states also ensures completeness. Lamport's construction has an advantage for finite-state programs, as the completed program retains its finite-state nature, but it is also rather drastic: ideally, a completion should expose only the information necessary for a proof.

Inspired by these completion proofs, this paper presents a fully automatic, gradual, *completion procedure* for finite-state programs. This differs from Lamport's construction by exposing predicates defined over local variables rather than the variables themselves, which can be more efficient. The starting point is a computation of the strongest split invariant. If the split invariant does not suffice to prove the property, local predicates are extracted from an analysis of error states contained in the current invariant, and added to the program as shared variables; then the computation is repeated. Unreachable error states are eliminated in successive rounds, while reachable error states are retained, and eventually detected.

Our algorithm is not optimal, as it does not always produce a minimal completion. It works well on a number of protocols, however, often showing a significant speedup over forward reachability. It is also useful in another setting, that of parameterized verification. In [26], it is observed that a quantified inductive invariant for a parameterized protocol induces a split invariant for each instance. In the other direction, under some restrictions, it is shown that it is possible to generalize a split invariant for a sufficiently large instance to a quantified inductive invariant which holds for all instances. Thus, stronger split invariants generated using the completion procedure can result in automatically generated proofs of correctness for the parameterized setting.

In summary, the main contributions of this paper are (i) a completion procedure for split invariance, and (ii) the experimental demonstration that, in many cases, the fixpoint calculation of split invariance, augmented with the completion method, works significantly better than forward reachability. Parameterized verification, while not the primary goal, is a welcome extra!

2 Background

This section defines split invariance and gives the fixpoint formulation of the strongest split invariant. Some of this material is taken from [26], it is repeated here for convenience.

Definition 0 *A* program *is given by a tuple* (V, I, T)*, where* V *is a set of (typed) variables,* $I(V)$ *is an initial condition, and* $T(V, V')$ *is a transition condition, where* V' *is a fresh set of variables in 1-1 correspondence with* V*.*

The semantics of a program is given by a *transition system*, which is a triple (S, S_0, R), where S is the state domain defined by the Cartesian product of the domains of variables in V, $S_0 = \{s : I(s)\}$, and $R = \{(s, t) : T(s, t)\}$. We assume that T is left-total, i.e., every state has a successor. A *state predicate* (also called an "assertion") is a Boolean expression over the program variables. The truth value of a predicate at a state is defined in the usual way by induction on formula structure. The expression $w(s)$ denotes the value of a variable w in state s.

Predicate Transformers We denote by *wlp* the weakest liberal precondition transformer introduced by Dijkstra [11]. For a predicate ξ, $wlp(M, \xi)$ denotes the weakest predicate (i.e., the largest set of states) from which all transitions of M lead to a state satisfying ξ. Namely,

$$wlp(M, \xi) = \{s \mid (\forall t : T(s, t) : \xi(t))\}$$

sp (also known as *post*) is the strongest post-condition, defined by

$$sp(M, \xi) = \{t \mid (\exists s : T(s, t) \wedge \xi(s))\}$$

Inductiveness and Invariance A state predicate φ is an *invariant* of program M if it holds at all reachable states of M. A state assertion ξ is an *inductive invariant* for M if it is initial (condition 1) and inductive (condition 2) (i.e., preserved by every program transition). The notation $[\psi]$, from Dijkstra and Scholten [13], indicates that ψ is valid.

$$[I_M \;\Rightarrow\; \xi] \tag{1}$$

$$[\xi \;\Rightarrow\; wlp(M, \xi)] \tag{2}$$

An inductive invariant is *adequate* to prove the invariance of a state predicate φ if it implies φ (condition 3).

$$[\xi \;\Rightarrow\; \varphi] \tag{3}$$

From the Galois connection between *wlp* and *sp*, condition 2 is equivalent to

$$[sp(M, \xi) \;\Rightarrow\; \xi] \tag{4}$$

The conjunction of conditions 1 and 4 is equivalent to

$$[(I_M \vee sp(M,\xi)) \Rightarrow \xi]$$

Since function $f(\xi) = I_M \vee sp(M,\xi)$ is monotonic, by the Knaster-Tarski theorem (below), it has a least fixpoint, which is the set of reachable states of M.

Theorem 0 *(Knaster-Tarski) A monotonic function f on a complete lattice has a least fixpoint, which is the strongest solution to $Z : [f(Z) \Rightarrow Z]$. Over* finite-height *lattices, it is the limit of the sequence $Z_0 = \bot$; $Z_{i+1} = f(Z_i)$.*

Program Composition The *asynchronous composition* of programs $\{P_i\}$, written as $(||i\colon P_i)$ is the program $P = (V, I, T)$, where the components are defined as follows. Let $V = (\bigcup i\colon V_i)$ and $I = (\bigwedge i\colon I_i)$. The transition condition T_i of program P_i is constrained so that it leaves local variables of other processes unchanged. Namely, define $\hat{T}_i$ as $T_i(V_i, V_i') \wedge (\forall j\colon j \neq i\colon \mathit{unchanged}(L_j))$, where $\mathit{unchanged}(X)$ means that the values of all variables in X are preserved from the current state to the next state. Then T is defined simply as $(\bigvee i\colon \hat{T}_i)$, and $\mathit{wlp}(P,\varphi)$ is equivalent to $(\bigwedge i\colon \mathit{wlp}(\hat{P}_i,\varphi))$, where $\hat{P}$ is the program P after modifying the transition relation to $\hat{T}_i$.

The *shared variables*, denoted X, are those that belong to $V_i \cap V_j$, for a distinct pair (i,j). The *local variables* of process P_i, denoted L_i, are the variables in V_i that are not shared (i.e., $L_i = V_i \setminus X$). The set of local variables is $L = (\bigcup i\colon L_i)$.

2.1 Split Invariance

For simplicity, we consider a two-process composition $P = P_1||P_2$; the results generalize to multiple processes. The desired invariance property φ is defined over the full product state of P. A *local* assertion for P_i is an assertion that is based only on V_i (equivalently, on X and L_i). A pair of local assertions $\theta = (\theta_1, \theta_2)$ is called a *split assertion*. Split assertion θ is a *split invariant* if the conjunction $\theta_1 \wedge \theta_2$ is an inductive invariant for P.

Split Invariance as a Fixpoint (Portions of this section are taken from [26] for completeness.)

The conditions for inductiveness of $\theta_1 \wedge \theta_2$ can be rewritten, following the Galois connection between *wlp* and *sp*, to the form below.

$$[sp(P_1, \theta_1 \wedge \theta_2) \Rightarrow (\theta_1 \wedge \theta_2)] \tag{5}$$
$$[sp(P_2, \theta_1 \wedge \theta_2) \Rightarrow (\theta_1 \wedge \theta_2)] \tag{6}$$

Re-arranging these in terms of θ_1 and θ_2, in combination with the initial condition gives the following implications, which are equivalent to the original. The

existential quantification in implications 7 (symmetrically for implications 8) over the local variables L_2 of P_2 does not lose equivalence, as these variables are irrelevant to θ_1 by the syntactic restriction on local assertions.

$$[(\exists L_2 : sp(P_1, \theta_1 \wedge \theta_2) \vee sp(P_2, \theta_1 \wedge \theta_2) \vee I) \Rightarrow \theta_1] \tag{7}$$
$$[(\exists L_1 : sp(P_1, \theta_1 \wedge \theta_2) \vee sp(P_2, \theta_1 \wedge \theta_2) \vee I) \Rightarrow \theta_2] \tag{8}$$

Implications 7 and 8, in turn, can be written as the pre-fixpoint formulation:

$$[\mathcal{F}(\theta_1, \theta_2) \preceq (\theta_1, \theta_2)]$$

where $\mathcal{F}$ is the pair function formed by the left-hand expressions in the implication, and $\preceq$ denotes pair-wise implication. Since $\mathcal{F}$ is monotone over (θ_1, θ_2) according to $\preceq$ (*sp* is a monotone function), by the Knaster-Tarski theorem, $\mathcal{F}$ has a *least* fixpoint.

The standard Knaster-Tarski algorithm for computing the least fixpoint by successive approximation results in the fixpoint solution (θ_1^*, θ_2^*), which is, by the derivation above, the *strongest* split invariant. The operations required to evaluate $\mathcal{F}$ (the computation of *sp*, and existential quantification) can be carried out by standard BDD manipulation for finite variable domains.

Theorem 1 *[26] A split invariance proof of the invariance of φ exists if, and only if, $[(\theta_1^* \wedge \theta_2^*) \Rightarrow \varphi]$.*

For a program with more than two processes, the general form of $\mathcal{F}_k(\theta)$ is

$$(\exists L \setminus L_k : I \vee (\bigvee j : sp(P_j, (\bigwedge m : \theta_m))))$$

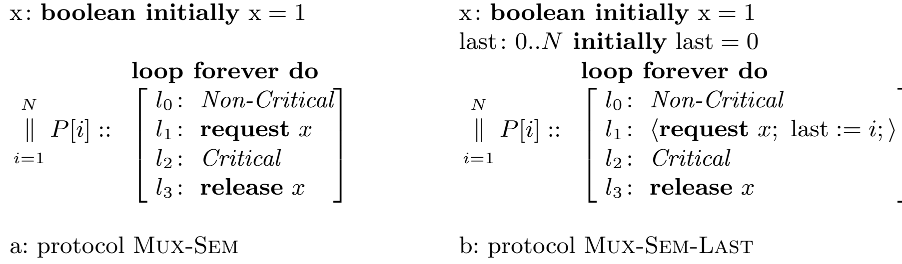
This definition can be read as follows: to compute the new value of θ_k using $\mathcal{F}_k(\theta)$, compute successors of θ with respect to each process P_j (the *sp* terms), add in the initial states, then project on to the variables X and L_k (equivalently, quantify out all local variables other than those in L_k).

3 The Full Procedure

The completeness problem, and its solution, is nicely illustrated by the mutual exclusion protocol in Figure 1.a, for which the safety property is

$$\forall i, j : i \neq j : \neg P[i].at_l_{2,3} \vee \neg P[j].at_l_{2,3}$$

Namely, for each pair of processes at least one is not in its critical section. (Note: the control predicate $P[i].at_l_{i,j}$ is an abbreviation of $P[i].at_l_i \vee P[i].at_l_j$.) For



a: protocol MUX-SEM b: protocol MUX-SEM-LAST

Fig. 1. Illustration of the (In)Completeness of Local Reasoning.

a 2-process instance, the strongest split invariant is *true*, i.e. all states are reachable. This includes (unreachable) states that violate mutual exclusion, making it impossible to prove the property. On the other hand, modifying the program by adding the auxiliary variable *last*, which records the last process to enter the critical section (Figure 1.b), results in the strongest split invariant given by

$$\theta = \bigwedge i\colon i \in \{1,2\}\colon\; P[i].at_l_{2,3} \;\equiv\; (\neg x \,\wedge\, \text{last} = i)$$

This suffices to prove mutual exclusion. The proof is by contradiction: if distinct processes P_i and P_j are in their critical sections together, the split invariant implies that last $= i$ and last $= j$, which is a contradiction. Our algorithm, *Split-Inv*, defined below, automatically discovers auxiliary variables such as this one.

A second route to completion, which we refer to as *Split-Inv-Pairwise*, is to widen the scope of local assertions to pairs of processes. A split invariant is now a matrix of entries of the form $\theta_{ij}(X, L_i, L_j)$. The 1-index fixpoint algorithm is extended to compute 2-index θ's as follows. Instead of n simultaneous equations, there are $O(n^2)$ equations, one for each pair (i,j) such that $i \neq j$. The operator, $\mathcal{F}_{ij}$, is defined as

$$(\exists L \setminus (L_i \,\cup\, L_j) : I \,\vee\, (\bigvee k\colon sp(P_k, \hat{\theta})))$$

where $\hat{\theta}$ is $(\bigwedge m\colon m \neq n\colon \theta_{mn})$. For the original program from Figure 1.a, *Split-Inv-Pairwise* produces the solution

$$\begin{aligned}\theta_{ij}(X, L_i, L_j) = \;& (x \;\Rightarrow\; (P[i].at_l_{0,1} \,\wedge\, \neg P[j].at_l_{2,3})) \;\wedge \\ & ((\neg x \,\wedge\, P[i].at_l_{2,3}) \;\Rightarrow\; \neg P[j].at_l_{2,3})\end{aligned}$$

which also suffices to prove mutual exclusion. Notice that no auxiliary variables are required. An interesting point is that, despite the quadratic number of calculations, pairwise split invariance outperformed both single-index split invariance (with completion) and reachability in some of our experiments.

3.1 Algorithm Split-Inv

The input to the algorithm is a concurrent program, P, with n processes, $\{P_i\}$, and a global property φ. There are two main steps. The initial step is to compute the strongest split invariant by the standard Knaster-Tarski iterative process, while checking that each stage has no errors. If this succeeds, the property is declared proved. If not, the refinement step either adds a new predicate, or enlarges the error set by adding some predecessors of the current error states. Two variables are updated during the algorithm: *pred* is the set of predicates; and *error* is the set of error states. This description gives the basic template, whereas Section 4 describes some of the heuristic variations that have been tried. In the description, we use θ^i to represent the i'th approximation $\theta^i_1 \wedge \theta^i_2 \wedge \ldots \wedge \theta^i_n$.

Definition 1 *Define states s and t to be* equivalent*, denoted $s \sim t$, if they have identical values for the original shared variables X, and the Boolean variables corresponding to the current set of predicates pred.*

Definition 2 *A* distinguishing pair *is a pair of states (s, t) such that $s \sim t$ but $error(s) \not\equiv error(t)$.*

Step 0 Initialize *pred* to $\emptyset$, and *error* to $\neg\varphi$
Step 1 If $(I \wedge error)$ is satisfiable, **HALT** with "The system fails to satisfy φ".
Step 2 Compute the split invariant for the system *augmented with the predicates pred* using the fixed point algorithm. If, at the $(i+1)$'st stage, $(\theta^{i+1} \wedge error)$ is satisfiable, proceed to Step 3. Otherwise, if a fixpoint is reached, **HALT** with "The system satisfies the property", and provide the split invariant as proof.
Step 3 Let $viol = \theta^{i+1} \wedge error$.
If there exists a distinguishing pair of states in θ^{i+1}, extract relevant predicates from this pair, add them to *pred*, and return to Step 1. Otherwise, continue to Step 4.
Step 4 Add the immediate predecessors of *viol* that are in θ^i to the error condition. I.e., modify *error* to $error \vee (\mathsf{EX}(viol) \wedge \theta^i)$. Return to Step 1.

The process for detecting new predicates, and augmenting the system to include the new predicates is described below. This is followed by an illustration of the algorithm on an example, and a proof of correctness.

3.2 Predicate Detection and Augmentation

As $\theta_1 \wedge \theta_2 \wedge ... \wedge \theta_n$ is always an over-approximation of the reachable states, *Split-Inv* may detect states that violate φ but are not actually reachable. Those states should be identified and left out of the split invariant. To do so, once a violating state is detected, *Split-Inv* computes essential predicates using a greedy strategy.

At Step 3, given a state s in *viol*, the greedy strategy tries to determine a state t such that (s, t) forms a distinguishing pair. For each *local* variable (from some process), the algorithm tests whether it is relevant to the error for that state; this is considered to be the case if an alternative value for the variable results in a non-error state in θ^{i+1}. If such a variant exists, the pair is a distinguishing pair.

For example, when the safety property is mutual exclusion, a violation occurs when two processes are in their critical section simultaneously. For such a violating state, the locations of these processes are essential for the error, and are represented as predicates (the location variable itself is then called an *essential variable*). The location of processes not in their critical section is irrelevant for the error at this state, so a predicate is not created for these processes.

The *augmentation* process works as follows. For each essential variable v of process P_i in an error state s, a predicate p of the form $(L_i = v(s))$ is added to *pred*, and a corresponding Boolean variable, b, is added to the shared variables of the program. It is initialized to the value of $p(L_i)$ at the initial state, and is updated as follows:

$$\begin{aligned} b' &\equiv p(L_i') && \text{for process } P_i \\ b' &\equiv b && \text{for process } P_j \text{ where } j \neq i \end{aligned}$$

This augmentation clearly does not affect the underlying transitions of the program: the new Boolean variables are purely auxiliary, and the transitions enforce the invariant $(b \equiv p(L_i))$.

Each component θ_i is now defined over X, L_i, and auxiliary Boolean variables. The auxiliary variables are used as additional constraints between θ_i and θ_j, sharpening the split invariant. A rough idea of how the sharpening works is as follows. Consider a state s to be "fixed" by the values of the auxiliary variables $b_1, \ldots, b_n$ (one for each process) if the local state components in s form the only satisfying assignment for $(\bigwedge i\colon b_i(s) \equiv p_i(L_i))$. The correctness proof shows (cf. Lemmas 1 and 2) that an unreachable error state with no predecessors is eliminated from the split invariance once it is fixed. However, a fixed, but unreachable, error state may be detected for the second time, if it has predecessors (which must be unreachable). In this case, the predecessors need to be eliminated, so they are considered as new error states.

Considering predecessors as error states and analyzing them in subsequent rounds continues until either sufficient new predicates are added for a proof, or the enlarged error set violates the initial condition. The latter indicates that the algorithm has found a real error since it detected a violating state and revealed a set of states that forms a valid path, beginning at an initial state and leading toward it.

It is important to point out that the algorithm is not necessarily optimal. When the algorithm detects a violating state it tries to expand predicates such that in

case the state does not have predecessors it will be eliminated in a subsequent round. The state, however, might have unreachable predecessors such that even after exposing the predicates it is detected in a following round. Only predicates that eliminate states with no predecessors cause unreachable violating states to be eliminated. Thus the algorithm might expose predicates which do not contribute to the elimination of unreachable violating states. It is conceivable that path-based analysis (as used in other abstraction-refinement approaches) may generate better predicates.

3.3 Illustration

We illustrate some of the key features of this algorithm on the Mux-Sem example from Figure 1.a. For simplicity we have only two processes; thus, the safety property is $\varphi \equiv \neg(P[1].at_l_{2,3} \wedge P[2].at_l_{2,3})$.

Iteration 0
Step 1 The initial condition is $x = 1 \wedge P[1].at_l_0 \wedge P[2].at_l_0$. It does not violate the safety property.
Step 2 *Split-Inv* computes the split invariant until $\theta_1 \wedge \theta_2$ violates φ. At this stage,

$$\begin{aligned} \theta_1 \wedge \theta_2 \equiv \quad & x = 0 \wedge P[1].at_l_{0,1,2} \wedge P[2].at_l_{0,1,2} \\ \vee\ & x = 1 \wedge P[1].at_l_{0,1} \quad \wedge P[2].at_l_{0,1} \end{aligned}$$

Step 3 Let *viol* be the set of states that satisfy $\theta_1 \wedge \theta_2 \wedge \neg\varphi$. The only state in *viol* is the one which satisfies $x = 0 \wedge P[1].at_l_2 \wedge P[2].at_l_2$, and together with the state that satisfies $x = 0 \wedge P[1].at_l_1 \wedge P[2].at_l_1$ they make a distinguishing pair. New essential predicates $P[1].at_l_2$ and $P[2].at_l_2$ are extracted, for which two new shared auxiliary variables b_1 and b_2 are added to the program, as described in Subsection 3.2. Since new predicates were found, a new iteration sets off in the next round.

Iteration 1
Step 1 The initial condition is $x = 1 \wedge P[1].at_l_0 \wedge P[2].at_l_0 \wedge \neg b_1 \wedge \neg b_2$, and again it does not violate the safety property.
Step 2 *Split-Inv* starts a new computation of the split invariant. Once again it is computed until $\theta_1 \wedge \theta_2$ violates φ. At this stage,

$$\begin{aligned} \theta_1 \wedge \theta_2 \equiv \quad & x = 0 \wedge P[1].at_l_{0,1,3} \wedge P[2].at_l_{0,1,3} \wedge \neg b_1 \wedge \neg b_2 \\ \vee\ & x = 0 \wedge P[1].at_l_{0,1} \quad \wedge P[2].at_l_2 \quad \wedge \neg b_1 \wedge b_2 \\ \vee\ & x = 0 \wedge P[1].at_l_2 \quad \wedge P[2].at_l_{0,1} \quad \wedge b_1 \wedge \neg b_2 \\ \vee\ & x = 1 \wedge P[1].at_l_{0,1} \quad \wedge P[2].at_l_{0,1} \quad \wedge \neg b_1 \wedge \neg b_2 \end{aligned}$$

Step 3 *viol* is computed, and again it consists of only one state which satisfies $x = 0 \wedge P[1].at_l_3 \wedge P[2].at_l_3$ (and makes a distinguishing pair together with $x = 0 \wedge P[1].at_l_1 \wedge P[2].at_l_1$). The shared variables b_3 and b_4, which are associated

with the essential predicates $P[1].at_l_3$ and $P[2].at_l_3$, respectively, are added to the program. A new iteration starts in the next round.

Iteration 2

Step 1 The initial condition is $x = 1 \wedge P[1].at_l_0 \wedge P[2].at_l_0 \wedge \neg b_1 \wedge \neg b_2 \wedge \neg b_3 \wedge \neg b_4$, and once again it does not violate the safety property.

Step 2 The split invariance is computed, however, this time a fixed point is reached. At this stage,

$$\begin{array}{rllllll} \theta_1 \wedge \theta_2 \equiv & x = 0 \wedge P[1].at_l_{0,1} & \wedge P[2].at_l_2 & \wedge \neg b_1 \wedge & b_2 \wedge \neg b_3 \wedge \neg b_4 \\ & \vee\, x = 0 \wedge P[1].at_l_{0,1} & \wedge P[2].at_l_3 & \wedge \neg b_1 \wedge \neg b_2 \wedge \neg b_3 \wedge & b_4 \\ & \vee\, x = 0 \wedge P[1].at_l_2 & \wedge P[2].at_l_{0,1} & \wedge \quad b_1 \wedge \neg b_2 \wedge \neg b_3 \wedge \neg b_4 \\ & \vee\, x = 0 \wedge P[1].at_l_3 & \wedge P[2].at_l_{0,1} & \wedge \neg b_1 \wedge \neg b_2 \wedge & b_3 \wedge \neg b_4 \\ & \vee\, x = 1 \wedge P[1].at_l_{0,1} & \wedge P[2].at_l_{0,1} & \wedge \neg b_1 \wedge \neg b_2 \wedge \neg b_3 \wedge \neg b_4 \end{array}$$

Since it does not violate φ the algorithm reports "The system satisfies the property", provides θ and halts.

3.4 Correctness

The correctness argument has to show that the procedure will eventually terminate, and detect correctly whether the property holds. For simplicity, the theorems are proved for the 2-process case, the proofs for the general case are similar. Lemmas 0, 1, and 2 show the effect that adding auxiliary Boolean variables has on subsequent split invariance calculations. Lemma 3 shows that a split invariant is always an over-approximation to the reachable states.

The state of a 2-process instance is represented using variable names X, b_1, b_2, L_1 and L_2, where X is the set of shared variables (of the original program), L_1, L_2 are the local variables of processes P_1, P_2 respectively, and b_1, b_2 are shared auxiliary Boolean variables added for predicates $p_1(L_1)$ and $p_2(L_2)$, respectively. Recall that, for a variable w, and a state s, $w(s)$ denotes the value of w in s.

Recall that states s and t are said to be equivalent, denoted $s \sim t$, if they have identical values for X, b_1, and b_2. A set of states S is closed under $\sim$ if, for each state in S, its equivalence class is included in S. A set of states is *pre-closed* if all predecessors of states in S are included in S.

Lemma 0 *(Invariance Lemma) The assertion* $(b_1 \equiv p_1) \wedge (b_2 \equiv p_2)$ *holds for all states in* $\theta_1^i \wedge \theta_2^i$*, for all approximation steps* i*.*

Proof. The proof is by induction. The claim holds trivially for $i = 0$, as the θ's are empty. Let s be a state in the $(i+1)$'st fixpoint approximation. As s is in θ_1^{i+1}, by the definition of $\mathcal{F}$, there is a variant state s' (which may differ from s in the value of L_2), that is either initial or is a successor of a state in $\theta_1^i \wedge \theta_2^i$. If s' is initial, then $b_1(s') \equiv p_1(L_1(s'))$ by the definition of the initial condition. If s' is a successor state, by the induction hypothesis, and the update expressions

for b_1 for a step of either P_1 or P_2, $b_1(s') \equiv p_1(L_1(s'))$. As s and s' agree on L_1 and b_1, this equivalence holds also for s. Moreover, as s is in θ_2^{i+1}, a symmetric proof shows that $b_2(s) \equiv p_2(L_2(s))$, establishing the claim. □

Lemma 1 *If state s is in the $(i+1)$'st approximation to the split invariant, there is an equivalent state t that is also in the $(i+1)$'st approximation, and either t is initial, or it has a predecessor in the i'th approximation.*

Proof. Let s be a state in the $(i+1)$'st approximation. By the definition of θ_1^{i+1}, s has a variant t, which may differ from it in the value of L_2, but agrees on the values of X,b_1, and b_2, such that t is either an initial state, or is a successor of a state in $\theta_1^i \wedge \theta_2^i$. From the update expressions for θ, t belongs to both θ_1^{i+1} and θ_2^{i+1}. □

The following lemma gives some insight into the split invariance procedure, and the effect of adding auxiliary variables. A variation of the argument is used in the proof of the main theorem.

Lemma 2 *(Exclusion Lemma) Let S be a set of states that is pre-closed, closed under $\sim$, and unreachable. Then S is excluded from the split invariant.*

Proof. The proof is by contradiction. Let $i+1$ be the first stage of the split-invariance calculation that contains a state from S, and let s be such a state. By Lemma 1, a state t equivalent to s must be in the $(i+1)$'st approximation, and t is either initial or has a predecessor in the previous approximation. As S is $\sim$-closed, t is in S, so it is unreachable, and cannot be an initial state. Thus, it must have a predecessor. This predecessor must be in S, as S is closed under taking predecessors. But this contradicts the assumption that $i+1$ is the first stage containing a state from S. □

Lemma 3 *(Reachability Lemma) The split invariant fixpoint is always an over-approximation of the set of reachable states.*

Proof. First, note that the addition of auxiliary Boolean variables does not affect the reachability of any state.

The claim follows from an easy induction on the split invariant calculation. The inductive hypothesis is that states that are reachable by a path with at most i states are included in the i'th approximation. This is true trivially for $i = 0$. Assuming true for i, consider a state s that is only reachable by a path of $i+1$ states. Then either $i = 0$ and s is initial, or $i > 0$ and s has an immediate predecessor on the path, t, which is in the i'th approximation. The update for θ_1^{i+1} is $(\exists L_2 : I \vee sp(P_1||P_2, \theta_1 \wedge \theta_2))$. The update expression ensures that s is in θ_1^{i+1}. A symmetric argument shows that s is in θ_2^{i+1}. □

Lemma 4 *It is an invariant of the algorithm that (i) $[\neg\varphi \Rightarrow error]$, and (ii) $[error \Rightarrow \mathsf{EF}(\neg\varphi)]$.*

Proof. The first claim follows from the initialization in Step 0, and the fact that *error* is only enlarged (in Step 4), and is unchanged otherwise. The second claim follows by induction from the initialization in Step 0, and the fact that the enlargement of *error* in Step 4 adds predecessors for some of the states in *error*. □

Theorem 2 *(Soundness) (a) If φ is declared to be proved, it is an invariant. (b) If φ is declared to fail, there is a reachable state where φ is false.*

Proof. Part (a): The contrapositive of Lemma 4(i) is $[\neg error \;\Rightarrow\; \varphi]$. Thus, if the split invariant has no error states, it implies φ. By Lemma 3, φ is true for all reachable states, and it is therefore invariant for the program.
Part (b): By Lemma 4 (ii), if an initial state satisfies *error*, there is a path to a state falsifying φ. □

Theorem 3 *The procedure always terminates.*

Proof. In this proof, we make key use of the fact that the programs are finite-state. Thus, there is a finite set of available predicates, call this $\mathcal{A}preds$. The progress measure is the pair $(|\mathcal{A}preds \setminus pred|, |\neg error|)$, i.e., the pair that measures the remaining available predicates, and the number of states that are not error states. Both quantities are non-negative, we order them by lexicographic order.

The measure is initialized to $(|\mathcal{A}preds|, |\varphi|)$ in Step 0. In Step 2 (the split-invariance calculation) the measure does not change . If Step 3 is enabled, a new predicate is added to *pred*; thus, the first component decreases strictly. Otherwise, in Step 4, the set of predicates stays constant, while states in $(\mathsf{EX}(viol) \wedge \theta^i)$ are added to *error*. We have to show that this set contains at least one state that is not already in *error*; if this is true, the second component of the measure decreases strictly.

Consider a state s in *viol*. By definition, it belongs to θ^{i+1} and *error*. As Step 3 is not applicable, there is no state t in θ^{i+1} such that (s,t) forms a distinguishing pair. Let t be the state guaranteed to exist by Lemma 1. Thus, t is equivalent to s, and in θ^{i+1}. Since (s,t) is not a distinguishing pair, t is also in *error*, and therefore in *viol*. By the lemma, t is either initial, or it has a predecessor in θ^i. The first case cannot hold, as the error (with t) would have been detected in Step 1. Thus, a predecessor t' of t is in $(\mathsf{EX}(viol) \wedge \theta^i)$, and t' cannot be in *error*, as θ^i is error-free. Thus, t' is a new state that is added to *error*. □

Theorem 4 *(Completeness I) If the property φ is an invariant for $P_1||P_2$, it is eventually proved.*

Proof. By Theorem 3, the algorithm terminates. By the contrapositive of Lemma 4(ii), $[\mathsf{AG}\varphi \;\Rightarrow\; \neg error]$. As φ is an invariant, this implies that all reachable states are not *error*-states. Thus, the test in Step 1 can never be true; hence, it must be that the algorithm terminates at Step 2 with the conclusion that the property is true, and the split invariant as proof. □

Theorem 5 *(Completeness II) If φ is not an invariant of $P_1||P_2$, this is eventually detected.*

Proof. By Theorem 3, the algorithm terminates. If the property is not invariant, there is a reachable state on which it fails. By the Reachability Lemma, the split invariant always includes these states; thus, the algorithm will not stop at Step 2 with success. Hence, it must terminate at Step 1 with an indication that the property does not hold. $\square$

4 Heuristics

Split-Inv as described in Subsection 3.1 is in its simplest form. In this section we provide several heuristics that may be used to speed up the computation. These heuristics are sound, but not necessarily complete.

Limiting Conjunction: In Owicki and Gries' deductive proof procedure, for each process P_i, one considers only the conjuncts $\theta_i \wedge \theta_j$, for $j \neq i$, rather than the full conjunction $\theta_1 \wedge \theta_2 \ldots \wedge \theta_n$. This is sound, but can result in a weaker split invariant. However, as fewer conjunctions are performed, this may speed up the split invariance calculation.

Early Quantification: Recall that, for a program with more than two processes, the general form of $\mathcal{F}_k(\theta)$ is

$$\exists L \setminus L_k \colon I \vee (\bigvee_j \colon sp(P_j, (\bigwedge_m \colon \theta_m)))$$

This expression may be optimized with early quantification, as follows. Distributing $\exists$ over $\vee$ and over sp, and using the fact that the θ_i's are local assertions, $\mathcal{F}_1(\theta)$ may be rewritten as follows, where lsp (read as "local sp") represents a quantified strongest postcondition.

$$(\exists L \setminus L_1 \colon I) \vee (\bigvee_j \colon lsp_1(P_j, \theta))$$

Similar expressions can be formulated for $\mathcal{F}_i$, for other values of i. This formulation quantifies out variables as early as possible. In this expression, $lsp_1(P_j, \theta)$ is defined as follows:

$$\begin{array}{ll} (\exists L_j \colon sp(P_j, \theta_1 \wedge \theta_j \wedge (\bigwedge k \colon k \notin \{1, j\} \colon (\exists L_k \colon \theta_k)))) & \text{for } j \neq 1 \\ sp(P_1, \theta_1 \wedge (\bigwedge k \colon k \neq 1 \colon (\exists L_k \colon \theta_k))) & \text{for } j = 1 \end{array}$$

Exploiting Symmetry in the Split Invariance Computation: Parameterized protocols are typically symmetric, i.e., the code of one process is isomorphic to the code of another, modulo the process index. It is shown in [26] that, as a result, the components θ_i and θ_j of the strongest split invariant are symmetric up to process index. This can be exploited during the calculation, instead of computing all the θ components, it suffices to compute only one, say θ_1, and derive the rest by substitution: $\theta_j = \theta_1[1 \leftarrow j]$. This scheme considerably reduces the number of computations, and is significantly faster than the full split-invariance computation.

Exploiting Symmetry in Refinement: Modify *Split-Inv* such that in step 3, after detecting a new predicate and adding the relevant auxiliary variable to the program, it should not further analyze states which satisfy the same predicate. This heuristic should only be applied for parameterized systems. The strength of this heuristic is very nicely illustrated by the following example: Consider again the example from Figure 1.a, instantiated with N equals to 4. The violating states $s_1\colon x = 0 \wedge P[1].at_l_3 \wedge P[2].at_l_3 \wedge P[3].at_l_0 \wedge P[4].at_l_0$ and $s_2\colon x = 0 \wedge P[1].at_l_0 \wedge P[2].at_l_3 \wedge P[3].at_l_3 \wedge P[4].at_l_0$ are detected in the same round. Assuming s_1 is analyzed first, relevant auxiliary variables are added for the predicates $P[1].at_l_3$ and $P[2].at_l_3$. According to the heuristic s_2 should not be analyzed during the same round because, like s_1, it satisfies $P[2].at_l_3$, thus the program is not going to be enhanced with an auxiliary variable for the predicate $P[2].at_l_3$ as a result of analyzing s_2. However, there are other violating states, for example, $s_3\colon x = 0 \wedge P[1].at_l_0 \wedge P[2].at_l_0 \wedge P[3].at_l_0 \wedge P[4].at_l_0$, which potentially may "contribute" that predicate. For parameterized systems, because of *Split-Inv*'s shape, once a violating state is detected, all its symmetric violating states are also detected in the same round. Sometimes this may cause *Split-Inv* to "skip over" an essential predicate instead of handling it; However, it is guaranteed that eventually all the unreachable violating states which satisfy this predicate are eliminated or else detected in a subsequent round. This heuristic appears to be very useful in reducing the running time by avoiding the analysis of violating states that do not contribute new essential predicates to the computation. on the other hand, it may lead to extra refinement computations.

Extracting Relevant Predicates: When analyzing a violating state in step 3, covering all combinations that cause the state to violate the safety property requires trying $2^n - 1$ possible permutations of subsets of processes. Instead, *Split-Inv* can be modified to chose processes eagerly and check the predicate of each process only once. If the predicate is not essential for the state to violate the safety property, as explained in Subsection 3.2, then it ignores that process (existentially quantify it) and continues analyzing the other processes. If it is essential, *Split-Inv* marks the predicate, keeps the state of its process (does not existentially quantify it) and continues analyzing the other processes. It is important to start this process by first analyzing predicates that were previously detected (if such exist) in order to detect new combinations of essential

predicates (if there are other combinations, analyzing predicates that were already detected first eliminates them thus revealing the other combinations). For example let's consider again the example from Figure 1.a, instantiated again with N equals 4. Observe the violating state $s_1 : x = 0 \wedge P[1].at_l_3 \wedge P[2].at_l_0 \wedge P[3].at_l_3 \wedge P[4].at_l_0$ (assuming no essential predicates were previously detected). Starting from the last process, the predicate $P[4].at_l_0$ is not essential for making s_1 a violating state. Therefore this predicate is existentially quantified and $s'_1 : x = 0 \wedge P[1].at_l_3 \wedge P[2].at_l_0 \wedge P[3].at_l_3$ is left to be analyzed. The predicate of the next process, $P[3].at_l_3$, is essential and therefore marked and kept. Next, $P[2].at_l_0$ is not essential and therefore it is existentially quantified, leaving $s''_1 : x = 0 \wedge P[1].at_l_3 \wedge P[3].at_l_3$ to be analyzed. $P[1].at_l_3$ is essential and therefore marked and kept.

BDD *variable ordering:* The program is augmented with new auxiliary variables in Step 3 of *Split-Inv*, if new essential predicates are detected. When using a symbolic model checker, which is based on BDD representation, it is sometimes more beneficial to add several shared Boolean variables before executing *Split-Inv*, without restricting them in the transition system (i.e. they may be changed nondeterministically in every transition). In Step 3, instead of adding new variables, these variables are used by modifying the transition relation and the initial condition as explained in Subsection 3.2. For some examples, this heuristic significantly reduces the size of the BDD and the running time of *Split-Inv*.

5 Experiments and Results

We implemented *Split-Inv* and *Split-Inv-Pairwise* using TLV [30], a BDD-based model checker, and tested it on protocols taken from the literature. The tests were conducted on a 2.8GHz Intel Xeon with 1GB RAM. For each experiment, the number of BDDs and the number of bytes encompass loading TLV and running the specific task. The running times only measure executing the specific task and do not include the time for loading TLV.

The primary aim of the experiments is to compare the two forms of our algorithm, i.e. *Split-Inv* and *Split-Inv-Pairwise*, against the forward reachability calculation on the full state space. *Split-Inv* is uniformly faster (sometimes significantly so) than forward reachability. Although not the main objective, we also compared our algorithms against model checking that uses inverse reachability (i.e., AG). The results were in favor of the latter for the MUX-SEM based examples, whereas both *Split-Inv* and *Split-Inv-Pairwise* obtained significantly better results for algorithm BAKERY, PETERSON'S and an incorrect mutual exclusion protocol.

For many protocols, including BAKERY and MUX-SEM, *Split-Inv* results in an inductive invariant that shows correctness for *all* instances, using the results in [26]. *Split-Inv* (as opposed to reachability) is essential for obtaining this result.

$$
\left[\begin{array}{lll}
\textbf{in} & N & : \textbf{natural where } N > 1 \\
\textbf{type} & Pr_id & : [1..N] \\
 & Level & : [0..N] \\
\textbf{local} & y & : \textbf{array } Pr_id \textbf{ of } Level \textbf{ where } y = 0 \\
 & s & : \textbf{array } Level \textbf{ of } Pr_id \\
\overset{N}{\underset{i=1}{\|}} & P[i] :: & \left[\begin{array}{l}
\textbf{loop forever do}: \\
\left[\begin{array}{l}
l_0: \ \textit{Non-Critical} \\
l_1: \ (y[i], s[1]) \ := (1, i) \\
l_2: \ \textbf{while } y[i] < N \textbf{ do} \\
\quad \left[\begin{array}{l}
l_3: \ \textbf{await } s[y[i]] \neq i \ \vee \ \forall j \neq i: \ y[j] < y[i] \\
l_4: \ (y[i], s[y[i]+1]) \ := (y[i]+1, i)
\end{array}\right] \\
l_5: \ \textit{Critical} \\
l_6: \ y[i] \ := 0
\end{array}\right]
\end{array}\right]
\end{array}\right]
$$

Fig. 2. PETERSON'S mutual exclusion protocol

As previously explained, *Split-Inv* consists of a loop with three main phases: computing the split invariant, refining the system by exposing predicates over local variables, and analyzing predecessors of violating states. It is important to point out that not all examples require the use of all three phases.

Method	Processes	BDDs	Bytes	Time(s)	Refinements	New Variables
Forward Reachability	2	2k	589k	0	-	-
Split-Inv	2	2k	589k	0	0	0
Split-Inv-Pairwise	2	1.8k	589k	0	0	0
Forward Reachability	5	23k	983k	0.06	-	-
Split-Inv	5	20k	917k	0.03	0	0
Split-Inv-Pairwise	5	26k	1M	0.08	0	0
Forward Reachability	10	195k	3.8M	0.95	-	-
Split-Inv	10	177k	3.5M	0.28	0	0
Split-Inv-Pairwise	10	266k	5.2M	1.9	0	0
Forward Reachability	20	1.8M	30M	126	-	-
Split-Inv	20	1.7M	29M	10.2	0	0
Split-Inv-Pairwise	20	3.1M	53M	281	0	0

Table 1. Test results for PETERSON'S mutual exclusion protocol.

The first example, provided in Figure 2, is PETERSON'S mutual exclusion protocol, for which the safety property requires that simultaneously not more than one process is in its critical section. Namely, $\forall i, j : i \neq j : (\neg P[i].at_l_{5,6} \vee \neg P[j].at_l_{5,6})$. *Split-Inv* terminates faster than forward model checking and *Split-Inv-Pairwise*, regardless the number of processes. It also appears that this example contain sufficient shared information for computing the split invariant, without having to employ any refinements. Table 1 provides the results for *Split-Inv*, *Split-Inv-Pairwise* and forward reachability.

Another algorithm that contains sufficient shared information for computing the split invariant without having to employ any refinements is BAKERY. The run

times achieved by *Split-Inv* are significantly better for larger instances. The detailed results are provided in Table 2. For both PETERSON'S mutual exclusion protocol and algorithm BAKERY, *Split-Inv* obtains better results than *Split-Inv-Pairwise* since none of them require any refinement steps, but *Split-Inv-Pairwise* includes quadratically more invariant computations.

Method	Processes	BDDs	Bytes	Time(s)	Refinements	New Variables
Forward Reachability	2	2.6k	590k	0	-	-
Split-Inv	2	2.5k	590k	0	0	0
Split-Inv-Pairwise	2	2.7k	590k	0	0	0
Forward Reachability	6	152k	3.1M	3.8	-	-
Split-Inv	6	139k	2.8M	1.6	0	0
Split-Inv-Pairwise	6	184k	3.8M	2	0	0
Forward Reachability	8	908k	15.2M	100	-	-
Split-Inv	8	783k	13.2M	28	0	0
Split-Inv-Pairwise	8	1.1M	18.8M	40.8	-	-
Forward Reachability	10	-	-	>2hrs	-	-
Split-Inv	10	4.5M	72M	992	0	0
Split-Inv-Pairwise	10	6.2M	102M	64min	0	0

Table 2. Test results for the algorithm BAKERY.

Another tested example was protocol MUX-SEM, which is a very simple solution for the mutual exclusion problem, where the coordination between the processes is done by a semaphore. The safety property of the protocol again requires that no computation contains a state in which two or more processes are in their critical section at the same time. A description of the protocol was provided in Figure 1 at the beginning of the paper. The run times of *Split-Inv* and *Split-Inv-Pairwise* are once again better than those of forward reachability. For this example *Split-Inv* requires two refinement steps in which new shared variables are added to the system in order to expose part of the local information. The detailed results for MUX-SEM are provided in Table 3.

$$\left[\begin{array}{lll} \textbf{in} & N & : \textbf{natural where } N > 1 \\ \textbf{local} & x & : \textbf{boolean where } x = 1 \\ \overset{N}{\underset{i=1}{\|}} & P[i] :: & \left[\begin{array}{l} \textbf{loop forever do}: \\ \quad \left[\begin{array}{ll} l_0: & \textit{Non-Critical} \\ l_1: & \textbf{request } x \\ l_2: & \textbf{release } x \end{array}\right] \end{array}\right] \end{array}\right]$$

Fig. 3. protocol MUX-SEM-SHORT

Protocol MUX-SEM-SHORT is a reduced version of MUX-SEM, in which location 3 is removed and the safety property is therefore $\forall i, j\colon i \neq j\colon (\neg P[i].at_l_2 \vee \neg P[j].at_l_2)$. Figure 3 provides a description of the protocol. For this protocol, regardless of the number of processes, *Split-Inv* requires only one refinement in

Method	Processes	BDDs	Bytes	Time(s)	Refinements	New Variables
Forward Reachability	2	0.3k	590k	0	-	-
Split-Inv	2	1k	590k	0	2	4
Split-Inv-Pairwise	2	0.3k	590k	0	0	0
Forward Reachability	10	10k	720k	0.03	-	-
Split-Inv	10	12k	852k	0.18	2	20
Split-Inv-Pairwise	10	10k	852k	0.05	0	0
Forward Reachability	20	19k	983k	0.96	-	-
Split-Inv	20	47k	16M	0.91	2	40
Split-Inv-Pairwise	20	24k	1.4M	0.29	0	0
Forward Reachability	50	116k	3M	23.3	-	-
Split-Inv	50	289k	6.4M	13.2	2	100
Split-Inv-Pairwise	50	190k	6.8M	4.4	0	0
Forward Reachability	100	462k	10M	405	-	-
Split-Inv	100	1.1M	24M	152	2	200
Split-Inv-Pairwise	100	1M	31M	130	0	0

Table 3. Test results for protocol Mux-Sem

which one bit of shared information is added for each process correspondingly. This bit indicates whether a process is in its critical section. Table 4 indicates that for this protocol *Split-Inv* obtains better run times than forward model checking.

Method	Processes	BDDs	Bytes	Time(s)	Refinements	New Variables
Forward Reachability	2	248	589k	0	-	-
Split-Inv	2	511	589k	0	1	2
Split-Inv-Pairwise	2	223	589k	0	0	0
Forward Reachability	20	19k	917k	0.86	-	-
Split-Inv	20	32k	1.1M	0.35	1	20
Split-Inv-Pairwise	20	22k	1.3M	0.22	0	0
Forward Reachability	50	114k	2.8M	21.4	-	-
Split-Inv	50	203k	4.5M	4.56	1	50
Split-Inv-Pairwise	50	184k	6.3M	3.36	0	0
Forward Reachability	100	459k	9.6M	384	-	-
Split-Inv	100	828k	16.3M	48	1	100
Split-Inv-Pairwise	100	1M	28M	101	0	0

Table 4. Test results for protocol Mux-Sem-Short

Protocol Mux-Sem-Last, presented in Figure 1 at the beginning of the paper, is a very simple enhancement of the Mux-Sem protocol, wherein a new shared variable, *last*, indicates which of the processes was most recently in its critical section. Immediately after a process enters its critical section, it modifies *last* to hold its ID. the safety property that is verified for this protocol is $\forall i, j\colon i \neq j\colon (\neg P[i].at_l_{2,3} \vee \neg P[j].at_l_{2,3})$. Table 5 indicates that for this protocol no refinements were required for computing the split invariant. When compared to forward model checking, *Split-Inv* produces better run times, smaller numbers of BDDs and less memory usage. *Split-Inv* also obtains better results than *Split-Inv-Pairwise* since, similarly to algorithm Bakery and Peterson's mu-

tual exclusion protocol, both do not require any refinement steps but *Split-Inv-Pairwise* includes quadratically more invariant computations.

Method	Processes	BDDs	Bytes	Time(s)	Refinements	New Variables
Forward Reachability	2	524	589k	0	-	-
Split-Inv	2	488	589k	0	0	0
Split-Inv-Pairwise	2	489	589k	0	0	0
Forward Reachability	10	10k	786k	0.16	-	-
Split-Inv	10	10k	720k	0.02	0	0
Split-Inv-Pairwise	10	10k	852k	0.08	0	0
Forward Reachability	20	47k	1.4M	2.79	-	-
Split-Inv	20	22k	983k	0.14	0	0
Split-Inv-Pairwise	20	36k	1.6M	0.57	0	0
Forward Reachability	50	449k	8.3M	207	-	-
Split-Inv	50	118k	2.8M	1.37	0	0
Split-Inv-Pairwise	50	298k	8.5M	19.6	0	0

Table 5. Test results for protocol MUX-SEM-LAST

A very interesting example is MUX-SEM-COUNT, of figure 4. This example is an enhancement of MUX-SEM where a counter is added locally for each process such that every time it enters into the critical section, the counter is increased by 1. The safety property is again $\forall i, j\colon i \neq j\colon (\neg P[i].at_l_{2,3} \vee \neg P[j].at_l_{2,3})$. Note, however, that it is independent of the counter.

$$
\left[\begin{array}{lll}
\textbf{in} & N, M & : \textbf{natural where } N, M > 1 \\
\textbf{local} & x & : \textbf{boolean where } x = 1 \\
\overset{N}{\underset{i=1}{\|}} & P[i] :: & \left[\begin{array}{l}
\textbf{local } \mathit{counter}\colon \textbf{integer where } \mathit{counter} = 0 \\
\textbf{loop forever do}\colon \\
\quad \left[\begin{array}{ll}
l_0\colon & \mathit{Non\text{-}Critical} \\
l_1\colon & \langle \textbf{request } x;\ \mathit{counter} := \mathit{counter} + 1\ (\textbf{mod } M)\rangle \\
l_2\colon & \mathit{Critical} \\
l_3\colon & \textbf{release } x
\end{array}\right]
\end{array}\right]
\end{array}\right]
$$

Fig. 4. protocol MUX-SEM-COUNT

The results for this example are again in favor of *Split-Inv* even that it always requires two refinement steps. When executing forward reachability it computes many similar states that only differ by the value of the counter, and even with relatively small values for M, having the counter leads to a state explosion very quickly. *Split-Inv* and *Split-Inv-Pairwise*, on the other hand, naturally abstract the value of the counter since it is never included as part of the essential predicates. Therefore, once an unreachable violating state is detected, after adding the relevant predicate variables, not only that the same state is eliminated in the following refinement step, but also other states, which differ from it only in the value of counter, are eliminated as well. The bound of the counter, i.e.

the value for M, does not really effects *Split-Inv* or *Split-Inv-Pairwise* while it has a crucial impact when computing forward reachability. Detailed results, for taking M equals to 10, are presented in Table 6.

Method	Processes	BDDs	Bytes	Time(s)	Refinements	New Variables
Forward Reachability	2	6.5k	720k	0	-	-
Split-Inv	2	8.9k	720k	0	2	4
Split-Inv-Pairwise	2	5.1k	720k	0	0	0
Forward Reachability	6	57k	1.7M	33	-	-
Split-Inv	6	19k	917k	0.93	2	12
Split-Inv-Pairwise	6	62k	1.9M	14.3	0	0
Forward Reachability	8	129k	3M	126	-	-
Split-Inv	8	32k	1.1M	1.87	2	16
Split-Inv-Pairwise	8	118k	3.1M	29.8	0	0
Forward Reachability	10	211k	4.5M	351	-	-
Split-Inv	10	49k	1.5M	3.31	2	20
Split-Inv-Pairwise	10	162k	4.2M	57	0	0

Table 6. Test results for protocol MUX-SEM-COUNT

TEST-AND-SET, is a variation of the example presented in [18], where the update to the shared variable is now defined to be the critical section. The protocol is shown in Figure 5 and the safety property that is verified is $\forall i, j : i \neq j : (\neg P[i].at_l_{2,3} \vee \neg P[j].at_l_{2,3})$. Table 7 shows that the TEST-AND-SET protocol requires several refinement steps in which the system is enhanced with more than one variable for each process.

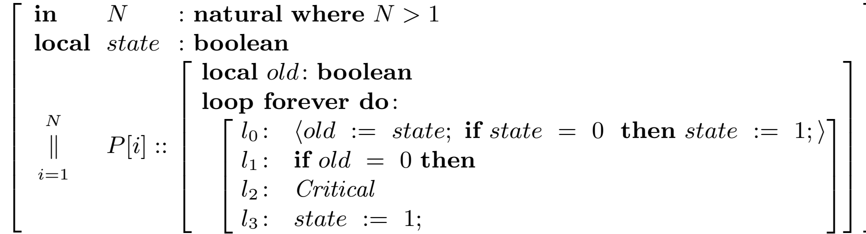


Fig. 5. Algorithm TEST-AND-SET

The obtained run times and the number of BDDs were this time in favor of forward model checking when compared to *Split-Inv*. However, when using *Split-Inv-Pairwise* the results turn over, and *Split-Inv-Pairwise* yields better run times (the number of BDDs and the memory usage were still better for model checking). For TEST-AND-SET, *Split-Inv* also results in an inductive invariant that shows correctness for all instances, using the results in [26]. It is also interesting to observe that when applying heuristic 3 of Section 4 the running time when

executing *Split-Inv* for the test case of 14 processes was reduced from over two hours to 6.4 seconds.

Method	Processes	BDDs	Bytes	Time(s)	Refinements	New Variables
Forward Reachability	2	592	589k	0	-	-
Split-Inv	2	2.4k	589k	0	2	6
Split-Inv-Pairwise	2	569	589k	0	0	0
Forward Reachability	6	8.5k	720k	0.01	-	-
Split-Inv	6	24k	1M	0.55	4	22
Split-Inv-Pairwise	6	7.3k	720k	0.02	0	0
Forward Reachability	10	10k	720k	0.23	-	-
Split-Inv	10	1M	17M	25.1	10	39
Split-Inv-Pairwise	10	10k	852k	0.1	0	0
Forward Reachability	14	20k	983k	0.75	-	-
Split-Inv	14	-	-	>2hrs	-	-
Split-Inv-Pairwise	14	17k	1.1M	0.24	0	0

Table 7. Test results for TEST-AND-SET.

All examples provided before were of correct protocols, i.e they all satisfy their safety properties. The next and last example, provided in Figure 6, is of an incorrect mutual exclusion protocol, MUX-SEM-TRY, and it illustrates the ability of *Split-Inv* to cope with systems that violate their own safety property and its ability to identify real violations. In this case, when performing the computation all three phases have to be employed, including several refinement steps in which multiple new variables are added and predecessors of violating states must be analyzed. The safety property that is verified is $\forall i, j \colon i \neq j \colon (\neg P[i].at_l_{2,3,4} \vee \neg P[j].at_l_{2,3,4})$.

$$
\left[\begin{array}{lll}
\textbf{in} & N & : \textbf{natural where } N > 1 \\
\textbf{local} & y & : \textbf{array } 1..N \\
\overset{N}{\underset{i=1}{\|}} & P[i] :: & \left[\begin{array}{l}
\textbf{loop forever do}: \\
\quad \left[\begin{array}{ll}
l_0 : & \textit{Non-Critical} \\
l_1 : & \textbf{await } \forall j \colon j \neq i \colon \neg y[j] \\
l_2 : & y[i] \ := 1 \\
l_3 : & \textit{Critical} \\
l_4 : & y[i] \ := 0
\end{array}\right]
\end{array}\right]
\end{array}\right]
$$

Fig. 6. protocol MUX-SEM-TRY

Table 8 compares forward reachability to *Split-Inv* for MUX-SEM-TRY. Both the number of BDDs and the run times achieved by *Split-Inv* and *Split-Inv-Pairwise* are significantly better. When performing tests on 20 processes, what requires more than 2 hours when using model checking is completed in 6.5 seconds when using *Split-Inv*, and we can only assume that as the number of processes increases - the difference increases as well.

Method	Processes	BDDs	Bytes	Time(s)	Refinements	New Variables
Forward Reachability	2	904	589k	0	-	-
Split-Inv	2	3.1k	589k	0	4	6
Forward Reachability	5	10k	720k	0.14	-	-
Split-Inv	5	10k	786k	0.09	4	12
Forward Reachability	10	337k	6M	28.4	-	-
Split-Inv	10	47k	1.4M	0.56	4	22
Forward Reachability	20	-	-	>2hrs	-	-
Split-Inv	20	365k	6.8M	6.5	4	42

Table 8. Test results for protocol Mux-Sem-Try

6 Related Work

Early work on compositional reasoning is primarily on deductive proof methods [10]. The pioneering methods of Owicki and Gries [27] (see also [12]) and Lamport [23] are extended to assume-guarantee reasoning by Chandy and Misra [3] and Jones [22], and to linear-time temporal properties (including liveness properties) by Pnueli [28]. The split invariance calculation can be viewed as mechanizing the Owicki-Gries proof rule, while the refinement procedure is inspired by Lamport's completeness proof.

Recent work on compositional reasoning is more algorithmic. Tools like Cadence SMV provide support for compositional proofs [25, 21]. Thread-modular reasoning [14, 15, 19] computes a per-process transition relation abstraction in a modular way. In [18], this abstraction is made more precise by including some aspects of the local states of other processes, and extended to parameterized verification. However, the method remains incomplete [15]. The abstraction algorithm of [24] avoids precision loss by preserving dependencies between local states, yet is also incomplete.

The split invariance computation is based on a simpler, state-based representation. The fixpoint algorithm for computing a strongest split invariant is by Namjoshi [26]. Cousot and Cousot also describe in [9] how non-interference arises from a conjunctive invariance formulation, but the paper does not provide a computation method. The key new aspect of this paper is that it addresses the central incompleteness problem.

The completion procedure is in the spirit of failure-based refinement methods, such as counter-example guided refinement [5]. Given a composition $P = P_1 || \ldots || P_n$, earlier refinement algorithms may be viewed as either abstracting P to a single process, which is successively refined; or applying compositional analysis to individual abstractions of each P_i. However, the latter method is incomplete, though compositional; while the first method is non-compositional, though complete. The procedure given here achieves both compositionality and completeness.

A different type of assume-guarantee reasoning applies machine learning to determine the weakest interface of a process as an automaton [17, 33, 16, 2]. It is complete, but the algorithms are complex, and may be expensive [7].

In [32], compositional verification and abstraction based on 3-valued games are combined into an automatic technique that can verify properties from the full μ-calculus. This differs from the present paper in that the technique operates on *synchronous* compositions, and has as a key step the global analysis of a composition of abstract processes.

Hu and Dill propose in [20] to dynamically partition the BDD's arising in a backward reachability (AG) computation. The partitioning is not necessarily local. A fixed local partitioning allows a simpler fixpoint procedure, and especially a simpler termination condition. Unlike split invariance, the Hu-Dill method computes the exact AG set. As the experiments show, however, over-approximation is not necessarily a disadvantage.

A split invariant, $(\wedge i : \theta_i)$, is quite similar in form to a quantified invariant, $(\forall i : \theta(i))$. This similarity is explored in [26], which shows a strong connection between split invariance and quantified invariants for parameterized protocols. A universally quantified invariant has the form $(\forall i : 0 \leq i < N : \theta(X, L_i))$, where N represents the size of an instance of the parameterized system. In one direction, a quantified inductive invariant must be invariant for each specific instance and, in fact, a split invariant for that instance, as the individual θ's are local assertions. (E.g., $\theta(X, L_0) \wedge \theta(X, L_1)$ is a split invariant for $N = 2$.) The converse is shown to hold in a restricted sense, under the assumption that the split invariant components are limited to assertions from a logic with a small model property. Under this restriction, a split invariant for a sufficiently large instance, viewed as a universally quantified formula, is an inductive invariant for all instances. The size of the "sufficiently large" instance is determined by the small model property for the logic. Thus, split invariance calculations over small instances can be used to construct parameterized invariants.

Logics with such properties are introduced in the context of the invisible invariants method [29]. This method generates quantified invariants using a heuristic approach, and can be used to show correctness for several of the protocols considered here. However, it is sometimes necessary to manually add auxiliary variables. It is unclear whether this is due to the limitations of the heuristics, or for a more fundamental reason.

The connection described above shows that the need is fundamental: as parameterized invariants correspond to split invariants, it is, in general, necessary to add auxiliary variables to obtain a sufficiently strong invariant. The completion procedure given here determines relevant variables for each specific instance of a parameterized system. For some protocols, the variables added for small instances suffice to construct a parameterized invariant. However, for other protocols, it is necessary to utilize existentially quantified auxiliary predicates (e.g., "there is at least one process in the critical region"). Developing a systematic, efficient, procedure to discover such predicates is an open question. The IIV tool [1], which implements the invisible invariants method, can sometimes discover such assertions through heuristics.

7 Conclusions and Future Work

This paper provides an algorithm—the first, to the best of our knowledge—which address the incompleteness problem for local reasoning. The local reasoning strategy itself computes a split state invariant, which is a simpler object than the transition relations or automata considered in other work.

Conceptually, local reasoning is an attractive alternative to model checking on the full state space. Our experiments show that this is justified in practice as well: split invariance, augmented with the completion procedure, can be a valuable model checking tool. In many cases, a split invariance proof can be used to show correctness of all instances of a parameterized protocol.

The completion procedure is defined for finite-state components. Extending this method to unbounded state components (e.g., C programs) would require a procedure that interleaves internal, per-process abstraction with split invariance and completion. Other interesting questions include exploring local reasoning for liveness properties (see [8]) and exploiting multi-core processing to speed up the independent θ_i computations in *Split-Inv*.

Acknowledgements This research was supported in part by the National Science Foundation under grant CCR-0341658. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

1. Ittai Balaban, Yi Fang, Amir Pnueli, and Lenore D. Zuck. IIV: An invisible invariant verifier. In *CAV*, volume 3576 of *LNCS*, pages 408–412, 2005.
2. S. Chaki, E. M. Clarke, N. Sinha, and P. Thati. Automated assume-guarantee reasoning for simulation conformance. In *CAV*, volume 3576 of *LNCS*, pages 534–547, 2005.
3. K.M. Chandy and J. Misra. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, 7(4), 1981.
4. E. M. Clarke and O. Grumberg. Avoiding the state explosion problem in temporal logic model checking. In *PODC*, pages 294–303, 1987.
5. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.
6. E.M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Workshop on Logics of Programs*, volume 131 of *LNCS*. Springer-Verlag, 1981.
7. 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 *ISSTA*, pages 97–108, 2006.
8. Ariel Cohen and Kedar S. Namjoshi. Local proofs for linear-time properties of concurrent programs. In *CAV*, 2008. To appear.

9. P. Cousot and R. Cousot. Invariance proof methods and analysis techniques for parallel programs. In A.W. Biermann, G. Guiho, and Y. Kodratoff, editors, *Automatic Program Construction Techniques*, chapter 12, pages 243–271. Macmillan, New York, New York, United States, 1984.
10. W-P. de Roever, F. de Boer, U. Hannemann, J. Hooman, Y. Lakhnech, M. Poel, and J. Zwiers. *Concurrency Verification: Introduction to Compositional and Noncompositional Proof Methods*. Cambridge University Press, 2001.
11. E.W. Dijkstra. *A Discipline of Programming*. Prentice Hall, Inc, 1976.
12. E.W. Dijkstra. EWD 554: A Personal Summary of the Gries-Owicki Theory. 1976. Available at `http://www.cs.utexas.edu/users/EWD`.
13. E.W. Dijkstra and C.S. Scholten. *Predicate Calculus and Program Semantics*. Springer Verlag, 1990.
14. C. Flanagan, S. N. Freund, S. Qadeer, and S. A. Seshia. Modular verification of multithreaded programs. *Theor. Comput. Sci.*, 338(1-3):153–183, 2005.
15. C. Flanagan and S. Qadeer. Thread-modular model checking. In *SPIN*, volume 2648 of *LNCS*, pages 213–224, 2003.
16. D. Giannakopoulou and C. S. Pasareanu. Learning-based assume-guarantee verification (tool paper). In *SPIN*, volume 3639 of *LNCS*, pages 282–287, 2005.
17. D. Giannakopoulou, C. S. Pasareanu, and H. Barringer. Assumption generation for software component verification. In *ASE*, pages 3–12, 2002.
18. T. A. Henzinger, R. Jhala, and R. Majumdar. Race checking by context inference. In *PLDI*, pages 1–13, 2004.
19. T. A. Henzinger, R. Jhala, R. Majumdar, and S. Qadeer. Thread-modular abstraction refinement. In *CAV*, volume 2725 of *LNCS*, pages 262–274, 2003.
20. A. J. Hu and D. L. Dill. Efficient verification with BDDs using implicitly conjoined invariants. In *CAV*, volume 697 of *LNCS*, pages 3–14, 1993.
21. R. Jhala and K. L. McMillan. Microarchitecture verification by compositional model checking. In *CAV*, volume 2102 of *LNCS*, pages 396–410. Springer, 2001.
22. C.B. Jones. *Development methods for computer programs including a notion of interference*. PhD thesis, Oxford University, 1981.
23. L. Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. Software Eng.*, 3(2), 1977.
24. Alexander Malkis, Andreas Podelski, and Andrey Rybalchenko. Precise thread-modular verification. In *SAS*, pages 218–232, 2007.
25. K.L. McMillan. A compositional rule for hardware design refinement. In *CAV*, volume 1254 of *LNCS*, 1997.
26. K. S. Namjoshi. Symmetry and completeness in the analysis of parameterized systems. In *VMCAI*, volume 4349 of *LNCS*, 2007.
27. S. S. Owicki and D. Gries. Verifying properties of parallel programs: An axiomatic approach. *Commun. ACM*, 19(5):279–285, 1976.
28. A. Pnueli. In transition from global to modular reasoning about programs. In *Logics and Models of Concurrent Systems*, NATO ASI Series, 1985.
29. A. Pnueli, S. Ruah, and L. D. Zuck. Automatic deductive verification with invisible invariants. In *TACAS*, volume 2031 of *LNCS*, pages 82–97, 2001.
30. A. Pnueli and E. Shahar. A platform for combining deductive with algorithmic verification. In *CAV*, volume 1102 of *LNCS*, pages 184–195, 1996. web: `www.cs.nyu.edu/acsys/tlv`.
31. J.P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proc. of the 5th International Symposium on Programming*, volume 137 of *LNCS*, 1982.

32. Sharon Shoham and Orna Grumberg. Compositional verification and 3-valued abstractions join forces. In *SAS*, pages 69–86, 2007.
33. O. Tkachuk, M. B. Dwyer, and C. S. Pasareanu. Automated environment generation for software model checking. In *ASE*, pages 116–129, 2003.