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 the concept of locality in proofs of global safety properties of asynchronously composed, multi-process 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 global safety property. Local proofs can be compact: but a central problem is that local reasoning is incomplete. In this paper, we present a "completion" algorithm, which gradually exposes facts about the internal state of components, until either a local proof or a real error is discovered. Experiments show that local reasoning can have significantly better performance over a reachability computation. Moreover, for some parameterized protocols, a local proof can be used to show correctness for *all* instances.

1 Introduction

The success achieved by model checking [5, 24] in various settings has always been tempered by the problem of state explosion [3]. 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. Many protocols for cache coherence and mutual exclusion, and multi-threaded programs, fit this program model. Other, more complex, safety properties can be reduced to invariance by standard methods.

Model checking tools typically 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 ensured 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 [20], it is shown that the *strongest* split invariant exists, and can be computed as a least fixpoint. Moreover, the split invariance formulation is nearly identical to the deductive proof method of Owicki and Gries [21] for compositional verification.

Intuitively, a local proof computation has advantages over a reachability computation. For one, each component of a split invariant can be expected to have a small BDD representation, as it is defined over the variables of a single process. Moreover, as the local assertions are loosely coupled—their only interaction is through the shared variables—BDD ordering constraints are less stringent.

On the other hand, a central problem with local reasoning is that it is incomplete: i.e., some valid properties do not have local proofs. This is because a split invariant generally over-approximates the set of reachable states, which may cause some unreachable error states to be included in the invariant. The over-approximation is due to the loose coupling between local states, as a joint constraint on L_i and L_j can be enforced only via X, by $\theta_i(X, L_i) \wedge \theta_j(X, L_j)$. Owicki and Gries showed that completeness can be achieved by adding auxiliary history variables to the shared state. Independently, Lamport showed in [18] that sharing all local state also ensures completeness. For finite-state processes, Lamport's construction has an advantage, 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.

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

The procedure is not optimal, in that it does not always produce a minimal completion. However, it works well on a number of protocols, often showing a significant speedup over forward reachability. It is also useful in another setting, that of parameterized verification. In [20], it is shown that split invariance proofs for small instances of a parameterized protocol can be generalized (assuming a small model property) to inductive invariants which show correctness of *all* instances. Completion helps in the creation of such proofs.

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!

An extended version of the paper, with complete proofs, and full experimental results, is available from `http://www.cs.bell-labs.com/who/kedar/local.html`.

2 Background

This section defines split invariance and gives the fixpoint formulation of the strongest split invariant. A more detailed exposition may be found in [20]. In the following, we assume that the reader is familiar with the concept of a state transition system.

Definition 0 *A component 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 (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* 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.

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 (1) and inductive (2) (i.e., preserved by every program transition). Here, *wlp* is the weakest liberal precondition transformer introduced by Dijkstra; the notation $[\psi]$, from Dijkstra and Scholten [8], indicates that ψ is valid.

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

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

An inductive assertion is *adequate* to show 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 the strongest post-condition operator *sp* (also known as *post*), condition (2) is equivalent to

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

The conjunction of (1) and (4) is equivalent to $[(I_M \,\vee\, sp(M, \xi)) \;\Rightarrow\; \xi]$. As 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 $(//i : P_i)$ is the program $P = (V, I, T)$, where the components are defined as follows. Let $V = (\cup i : V_i)$, and $I = (\wedge i : I_i)$. 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 $L = (\cup i : L_i)$. The transition condition T_i of program P_i is constrained so that it leaves local variables of other processes unchanged. I.e., T_i is extended to $T_i(V_i, V_i') \wedge (\forall j : j \neq i : L_j' = L_j)$. Then T can be defined simply as $(\vee i : T_i)$, and $wlp(P, \varphi)$ is equivalent to $(\wedge i : wlp(P_i, \varphi))$.

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 The conditions for inductiveness of $\theta_1 \wedge \theta_2$ can be rewritten to the simultaneous pre-fixpoint form below, based on the (sp, wlp) Galois connection and locality. In particular, the existential quantification over local variables encodes locality, as θ_i is independent of L_j, for $j \neq i$.

$$[(\exists L_2 : I \vee sp(P_1, \theta_1 \wedge \theta_2) \vee sp(P_2, \theta_1 \wedge \theta_2)) \Rightarrow \theta_1] \tag{5}$$

$$[(\exists L_1 : I \vee sp(P_1, \theta_1 \wedge \theta_2) \vee sp(P_2, \theta_1 \wedge \theta_2)) \Rightarrow \theta_2] \tag{6}$$

Let $\mathcal{F}_i(\theta)$ refer to the left-hand side of the implication for θ_i. By monotonicity of $\mathcal{F}_i$ in terms of (θ_1, θ_2) and the Knaster-Tarski theorem, there is a strongest solution, θ^*, which is also a simultaneous least fixpoint: $[\theta_i^* \equiv \mathcal{F}_i(\theta^*)]$. For finite-state programs, $\mathcal{F}_i(\theta)$ can be evaluated using standard BDD operations.

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

Early Quantification For a program with more than two processes, the general form of $\mathcal{F}_1(\theta)$ is $(\exists L \setminus L_1 : I \vee (\vee j : sp(P_j, (\wedge m : \theta_m))))$. 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 to $(\exists L \setminus L_1 : I) \vee (\vee j : lsp_1(P_j, \theta))$, which quantifies out variables as early as possible. In this expression, $lsp_1(P_j, \theta)$ is defined as follows: for $j \neq 1$, it is $(\exists L_j : sp(P_j, \theta_1 \wedge \theta_j \wedge (\wedge k : k \notin \{1, j\} : (\exists L_k : \theta_k))))$, and for $j = 1$, it is $sp(P_1, \theta_1 \wedge (\wedge k : k \neq 1 : (\exists L_k : \theta_k)))$.

```
x: boolean initially x = 1

                    loop forever do
                    ⎡ I: Non-Critical ⎤
 $\|_{i=1}^{N}$ P[i] ::  ⎢ T: request x     ⎥
                    ⎢                  ⎥
                    ⎢ C: Critical      ⎥
                    ⎣ E: release x     ⎦
```

(a) protocol MUX-SEM

```
x: boolean initially x = 1
last: 0..N initially last = 0
                    loop forever do
                    ⎡ I : Non-Critical ⎤
 $\|_{i=1}^{N}$ P[i] ::  ⎢ T : request x     ⎥
                    ⎢     last := i;    ⎥
                    ⎢ C : Critical      ⎥
                    ⎣ E : release x     ⎦
```

(b) protocol MUX-SEM-LAST

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

3 The Completion Procedure

The completeness problem, and its solution, is nicely illustrated by the mutual exclusion protocol in Figure 1(a). For a 2-process instance, the strongest split invariant is $(\mathit{true}, \mathit{true})$. 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 $\theta_i = ((C_i \vee E_i) \equiv ((\neg x) \wedge \text{last} = i))$. This suffices to prove mutual exclusion. The completion algorithm, COMPLETION, defined below, automatically discovers auxiliary variables such as this one.

A second route to completion, which we refer to as COMPLETION-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 $(\exists L \setminus (L_i \cup L_j) : I \vee (\vee k : sp(P_k, \hat{\theta})))$, where $\hat{\theta}$ is $(\wedge m, n : m \neq n : \theta_{mn})$. For the original program from Figure 1(a), COMPLETION-PAIRWISE produces the solution $\theta_{ij}(X, L_i, L_j) = ((x \Rightarrow ((I_i \vee T_i) \wedge \neg C_j)) \wedge ((\neg x \wedge C_i) \Rightarrow \neg C_j))$, which suffices to prove mutual exclusion. It is interesting that, in some of our experiments, pairwise split invariance outperformed both single-index split invariance (with completion) and reachability.

3.1 The Completion Algorithm

We first provide a description of the main steps of the algorithm COMPLETION. The input is a concurrent program, P, with n processes, $\{P_i\}$, and a global property φ. We use θ^i to represent the i'th approximation $\theta^i_1 \wedge \theta^i_2 \wedge \ldots \wedge \theta^i_n$. The *refinement phase* (steps 3 and 4) can be optimized without violating the correctness argument; this is discussed in the extended version of the paper.

1. If the initial condition violates φ, halt with "Error".
2. Compute the split invariant using the fixed point algorithm. If, at the i'th stage, θ^i violates φ, go to step 3. If a fixpoint is reached, halt with "Verified" and provide the split invariant as proof.

3. Let *viol* $= \theta^i \wedge \neg\varphi$. For each state in *viol*, find new *essential predicates* and add auxiliary variables for these to the program. If new predicates are found, return to step 1, which starts a new split invariance calculation; otherwise, continue to step 4.
4. Add the immediate predecessors of *viol* to the error condition—i.e., modify φ to $\varphi \wedge \neg\mathsf{EX}(\mathit{viol})$—and return to step 3.

3.2 The Refinement Phase

As $\theta_1 \wedge \theta_2 \wedge ... \wedge \theta_n$ is always an over-approximation of the reachable states, COMPLETION 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, COMPLETION computes essential predicates using a greedy strategy. 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. (Sometimes, a group of variables may need to be considered together.) For example, mutual exclusion is violated for a global state if two processes are at the critical location, but the locations of other processes are not relevant, since they could be set to arbitrary values while retaining the error condition.

For each relevant variable v in an error state s, a predicate of the form $v = v(s)$ is added to the program. This is a local predicate, as v is a local variable for some process. To add a predicate $f(L_i)$, a corresponding Boolean variable b is added to the shared state, and initialized to the value of $f(L_i)$ at the initial state. It is updated as follows: for process P_i, the update is given by $b' \equiv f(L_i')$, and for process P_j, $j \neq i$, the update is given by $b' \equiv b$. 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 $(b \equiv f(L_i))$.

Each component θ_i is now defined over X, L_i, and the auxiliary Boolean variables. The auxiliary variables act as additional constraints between θ_i and θ_j, sharpening the split invariant. A rough idea of how the sharpening works is as follows. (A precise formulation is in Section 3.4.) 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 $(\wedge i : b_i(s) \equiv f_i(L_i))$. 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 error states by modifying φ, and predicates are extracted from them.

Adding predecessors continues until (i) at least one new predicate is exposed, and a new computation is initialized, or (ii) the modified φ violates the initial condition – an indication that a state violating the original φ is reachable.

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 $\varphi \equiv \neg(C_1 \wedge C_2)$.

Iteration 0
Step 1 The initial condition is $x = 1 \wedge I_1 \wedge I_2$. φ does not violate it.
Step 2 COMPLETION computes the split invariant until $\theta_1 \wedge \theta_2$ violates φ. At this stage,

$$\begin{aligned} \theta_1 \wedge \theta_2 \equiv \quad & x = 1 \wedge ((I_1 \vee T_1) \wedge (I_2 \vee T_2)) \\ \vee\, & x = 0 \wedge ((I_1 \vee T_1 \vee C_1) \wedge (I_2 \vee T_2 \vee C_2)) \end{aligned}$$

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 $x = 0 \wedge C_1 \wedge C_2$. The global predicate variables b_1 and b_2, which are associated with the essential predicates C_1 and C_2, respectively, are added to the program, as described previously.

Iteration 1
Step 2 A new computation of the split invariant sets off. Once again it is computed until $\theta_1 \wedge \theta_2$ violates φ. The description of $\theta_1 \wedge \theta_2$ is long, and is omitted, but the important point is that $x = 0 \wedge C_1 \wedge C_2 \wedge b_1 \wedge b_2$ satisfies it.
Step 3 Since $(x = 0, C_1, C_2)$ was already detected, the negations of its predecessors that satisfy $\theta_1 \wedge \theta_2$ are added to φ, i.e. φ is augmented by $\neg(x = 1 \wedge C_1 \wedge T_2)$ and $\neg(x = 1 \wedge T_1 \wedge C_2)$ and the corresponding states are analyzed as well. Since both predecessors satisfy $\theta_1 \wedge \theta_2$, violate φ, and are detected for the first time, new global predicate variables b_3 and b_4, which are associated with the essential predicates T_2 and T_1, respectively, are added to the program.

Iteration 2
Again, the split invariance calculation does not succeed. This time, the error states $(x = 1, C_1, T_2, b_1, \neg b_2, b_3, \neg b_4)$ and $(x = 1, T_1, C_2, \neg b_1, b_2, \neg b_3, b_4)$ are part of the split invariant.
Step 3 Since both of these states were already detected, the negations of their predecessors that belong to $\theta_1 \wedge \theta_2$ are added to φ, i.e. φ is augmented by $\neg(x = 1 \wedge C_1 \wedge I_2)$ and $\neg(x = 1 \wedge I_1 \wedge C_2)$, and they are analyzed as well. Since both predecessors belong to $\theta_1 \wedge \theta_2$, violate φ, and are detected for the first time, new global predicate variables b_5 and b_6, which are associated with the essential predicates I_2 and I_1, respectively, are added to the program.

Iteration 3
The split invariance calculation succeeds, establishing mutual exclusion.

3.4 Correctness

The correctness argument has to show that the procedure will eventually terminate, and detect correctly whether the property holds. The theorems are proved for the 2-process case, the proof for the general case is similar. Lemmas 0, 1, and 2 make precise 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.

To represent the state of a 2-process instance, we use variables X, b_1, b_2, L_1, L_2, where X represents the shared variables, L_1, L_2 are the local variables of processes P_1, P_2 respectively, and b_1, b_2 are auxiliary Boolean variables added for predicates $f_1(L_1)$ and $f_2(L_2)$, respectively. For a variable w, and a state s, let $w(s)$ denote the value of w in s.

Define states s and t to be *equivalent*, denoted $s \sim t$, if they agree on the 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* $(b_1 \equiv f_1) \wedge (b_2 \equiv f_2)$ *holds for all states in* $\theta_1^i \wedge \theta_2^i$*, for all approximation steps* i*.*

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.*

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.*

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

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): If the split invariant implies φ, by Lemma 3, φ is true of all reachable states, and is therefore invariant.
Part (b): This follows as the error states, which are initially a subset of $\neg\varphi$, are enlarged by adding predecessors. Thus, if an initial state is considered to be an error, there is a path to a state falsifying φ. □

Theorem 3 *(Completeness I) If the property* φ *is an invariant for* $P_1//P_2$*, it is eventually proved.*

Proof. If φ is an invariant, any states in the first split invariant that do not satisfy φ are unreachable. Call this set *error*. The procedure used to add predicates (steps 3 and 4), in the limit, extracts predicates from all states in $\mathsf{EF}(\mathit{error})$, as it adds predecessors to the error set. The set $\mathsf{EF}(\mathit{error})$ is pre-closed, and unreachable. If this set is not $\sim$-closed, there are states s and t such that $s \sim t$, but s is an error state, while t is not—this triggers the addition of a new predicate in Step 3 of the algorithm. As there are only finitely many predicates, eventually, enough predicates are added so that the set is $\sim$-closed. By Lemma 2, once $\sim$-closure is obtained, the set is excluded from the split invariant. At this stage, the split invariant has no error states, and the property is declared proved. □

Theorem 4 *(Completeness II) If φ is not an invariant of $P_1//P_2$, this is eventually detected.*

Proof. 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. The completion procedure, at each step, will enlarge the error set, effectively computing $\mathsf{EF}(\mathit{error})$. At some stage (defined by the length of the shortest path to an error state) this has a non-empty intersection with the initial states, at which point the error is detected. □

Theorems 3 and 4 also show termination of the procedure.

4 Experiments and Results

We implemented COMPLETION using TLV [23], 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.

The primary aim of the experiments is to compare split invariance with the two forms of completion against a forward reachability calculation on the full state space. The split invariance calculation is uniformly faster (sometimes significantly so) than forward reachability. We also compared it against model checking using inverse reachability (i.e., AG). In three examples (PETERSON'S, BAKERY, and an incorrect mutual exclusion protocol), split invariance performs significantly better than the AG calculation; in other examples, the AG calculation is somewhat faster.

For many of these protocols, including BAKERY and MUX-SEM, the split invariance calculation also results in an inductive invariant that shows correctness for *all* instances, using the results in [20]. Split invariance (as opposed to reachability) is essential for obtaining this result.

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

For two examples: PETERSON'S mutual exclusion protocol and algorithm BAKERY, COMPLETION terminated much faster than traditional forward or backward model checking. It appears that these examples contain sufficient global information for computing the split invariant, without having to employ any refinements. Table 1 compares COMPLETION, forward reachability and inverse reachability for

Method	Processes	BDDs	Bytes	Time(s)	Refinements	New Variables
Forward Reachability	2	2k	524k	0	-	-
Backward Reachability	2	1.7k	524k	0	-	-
COMPLETION	2	2k	524k	0	0	0
Forward Reachability	5	23k	917k	0.05	-	-
Backward Reachability	5	42k	1.2M	0.29	-	-
COMPLETION	5	20k	852k	0.04	0	0
Forward Reachability	10	194k	3.7M	0.94	-	-
Backward Reachability	10	13M	211M	680	-	-
COMPLETION	10	173k	3.4M	0.26	0	0
Forward Reachability	20	1.8M	30M	127	-	-
Backward Reachability	20	-	-	>2hrs	-	-
COMPLETION	20	1.7M	29M	9.9	0	0

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

PETERSON'S mutual exclusion protocol. The run times achieved by COMPLETION are significantly better for larger instances.

Another tested example was protocol MUX-SEM, provided in Figure 1. When running COMPLETION in its basic form, the obtained run times and the number of BDDs were not as good as those of traditional forward model checking, due to the overhead of the multiple split invariance runs. However, when we use a pairwise split invariant computation, as explained in the introduction of Section 3, the results turn over, and the run times are in COMPLETION's favor. Backward reachability obtained the best results for this protocol.

All examples provided before were of correct protocols, i.e they all satisfied their safety properties. The next and last example is of an incorrect mutual exclusion protocol, MUX-SEM-TRY, and it illustrates the ability of COMPLETION 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 had to be employed, together with several refinements in which multiple new variables where added and the predecessors of violating states had to be analyzed.

Table 2 compares forward and backward reachability to COMPLETION for MUX-SEM-TRY. Both the number of BDDs and the run times achieved by COMPLETION are significantly better. When performing tests on 20 processes, what requires more than 2 hours when using model checking is completed in 52 seconds when using COMPLETION, and we can only assume that as the number of processes increases - the difference increases as well.

5 Related Work

Early work on compositional reasoning is primarily on deductive proof methods [7]. The pioneering methods of Owicki and Gries [21] and Lamport [18] are extended to assume-guarantee reasoning by Chandy and Misra [2] and Jones [17]. The split invariance calculation can be viewed as mechanizing the Owicki-Gries proof rule, while the completion algorithm is inspired by Lamport's method.

Method	Processes	BDDs	Bytes	Time(s)	Refinements	New Variables
Forward Reachability	2	877	524k	0	-	-
Backward Reachability	2	1.1k	524k	0	-	-
COMPLETION	2	4.8k	589k	0.01	5	8
Forward Reachability	5	11k	720k	0.12	-	-
Backward Reachability	5	10k	655k	0.26	-	-
COMPLETION	5	10k	720k	0.16	7	14
Forward Reachability	10	337k	6M	27.7	-	-
Backward Reachability	10	450k	7.8M	25.8	-	-
COMPLETION	10	70k	1.7M	1.3	7	24
Forward Reachability	20	-	-	>2hrs	-	-
Backward Reachability	20	-	-	>2hrs	-	-
COMPLETION	20	1M	18M	35	7	44

Table 2. Test results for protocol MUX-SEM-TRY

Recent work on compositional reasoning is more algorithmic. Tools like Cadence SMV provide support for compositional proofs [19, 16]. “Thread-modular” reasoning [9, 10, 14] computes a per-process transition relation abstraction in a modular way. In [13], this abstraction is made more precise by including some aspects of the local states of other processes, and extended to parameterized verification.

Split invariance is based on a simpler, state-based representation. A key new aspect of this paper is that it addresses the central incompleteness problem. While a transition relation abstraction is more precise than one based on states, it is incomplete nonetheless [10].

The “invisible invariants” method [22] heuristically generates quantified invariants for parameterized protocols. This can prove correctness for many of the protocols considered here, but it requires the user-guided addition of auxiliary variables in several cases. One of the contributions of this paper is to automate the addition of such auxiliaries.

The completion procedure is in the spirit of failure-based refinement methods, such as counter-example guided refinement [4]. Given a composition $P = P_1 // \ldots // P_n$, earlier refinement algorithms may be viewed as either (1) abstracting P to a single process, which is successively refined; or (2) applying compositional analysis to individual abstractions of each P_i. However, method (2) is incomplete, though compositional; while method (1) 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 [12, 25, 11, 1]. This is complete, but the algorithms are complex, and may be expensive [6].

Hu and Dill propose in [15] to dynamically partition the BDD’s arising in a reachability 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 set of reachable states. As the experiments show, however, over-approximation is not necessarily a disadvantage.

6 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 the design of a split invariance procedure for synchronous composition, and the investigation of local reasoning for liveness properties.

Acknowledgements This research was supported, in part, by NSF grant CCR-0341658.

References

1. 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.
2. K.M. Chandy and J. Misra. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, 7(4), 1981.
3. E. M. Clarke and O. Grumberg. Avoiding the state explosion problem in temporal logic model checking. In *PODC*, pages 294–303, 1987.
4. 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.
5. 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.
6. 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.
7. W-P. de Roever, F. de Boer, U. Hannemann, J. Hooman, Y. Lakhnech, M. Poel, and J. Zwiers. *Concurrency Verification: Introduction to Compositional and Non-compositional Proof Methods*. Cambridge University Press, 2001.
8. E.W. Dijkstra and C.S. Scholten. *Predicate Calculus and Program Semantics.* Springer Verlag, 1990.
9. 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.
10. C. Flanagan and S. Qadeer. Thread-modular model checking. In *SPIN*, volume 2648 of *LNCS*, pages 213–224, 2003.

11. D. Giannakopoulou and C. S. Pasareanu. Learning-based assume-guarantee verification (tool paper). In *SPIN*, volume 3639 of *LNCS*, pages 282–287, 2005.
12. D. Giannakopoulou, C. S. Pasareanu, and H. Barringer. Assumption generation for software component verification. In *ASE*, pages 3–12, 2002.
13. T. A. Henzinger, R. Jhala, and R. Majumdar. Race checking by context inference. In *PLDI*, pages 1–13, 2004.
14. T. A. Henzinger, R. Jhala, R. Majumdar, and S. Qadeer. Thread-modular abstraction refinement. In *CAV*, volume 2725 of *LNCS*, pages 262–274, 2003.
15. 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.
16. R. Jhala and K. L. McMillan. Microarchitecture verification by compositional model checking. In *CAV*, volume 2102 of *LNCS*, pages 396–410. Springer, 2001.
17. C.B. Jones. *Development methods for computer programs including a notion of interference*. PhD thesis, Oxford University, 1981.
18. L. Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. Software Eng.*, 3(2), 1977.
19. K.L. McMillan. A compositional rule for hardware design refinement. In *CAV*, volume 1254 of *LNCS*, 1997.
20. K. S. Namjoshi. Symmetry and completeness in the analysis of parameterized systems. In *VMCAI*, volume 4349 of *LNCS*, 2007.
21. S. S. Owicki and D. Gries. Verifying properties of parallel programs: An axiomatic approach. *Commun. ACM*, 19(5):279–285, 1976.
22. A. Pnueli, S. Ruah, and L. D. Zuck. Automatic deductive verification with invisible invariants. In *TACAS*, volume 2031 of *LNCS*, pages 82–97, 2001.
23. 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`.
24. 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.
25. O. Tkachuk, M. B. Dwyer, and C. S. Pasareanu. Automated environment generation for software model checking. In *ASE*, pages 116–129, 2003.