Local Proofs for Linear-Time Properties of Concurrent Programs

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

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

Abstract. This paper develops a local reasoning method to check linear-time temporal properties of concurrent programs. In practice, it is often infeasible to model check over the product state space of a concurrent program. The method developed in this paper replaces such global reasoning with checks of (abstracted) individual processes. An automatic refinement step gradually exposes local state if necessary, ensuring that the method is complete. Experiments with a prototype implementation show that local reasoning can hold a significant advantage over global reasoning.

1 Introduction

Model Checking [5, 34] has been singularly successful at automating (in)correctness proofs of programs. On the other hand, the standard model checking method suffers from a serious *state explosion* problem [6]. For concurrent programs, state explosion is caused by an exponential growth of the global state space with increasing number of components. It is often necessary to use abstraction and compositional reasoning methods to break up a model checking question into a series of local questions.

This paper develops just such a *local reasoning* method, for analyzing linear temporal properties of asynchronously composed, concurrent programs. In the first step, a *split invariant* (a conjunction of local, per-process invariants) is calculated, as shown in [30, 9]. The next step is to construct abstractions of individual processes, based on the split invariant. Finally, liveness properties are checked individually on each abstraction. A contribution of this work is the *derivation* of this method from a deductive reasoning rule similar to that of Owicki and Gries [31]. The deductive rule is based on user-supplied rank functions: the derivation shows how to replace these with model checking.

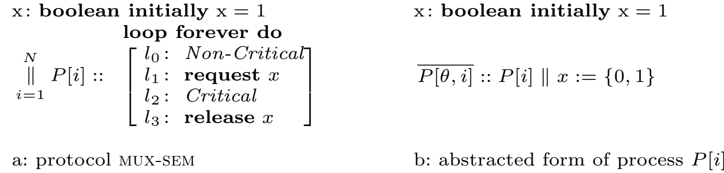
Local reasoning is inherently incomplete: informally speaking, each process abstraction can view only part of the behavior of the other processes, which may not suffice to establish a property. Both Owicki and Gries, and Lamport [28] propose ways to resolve this problem. Owicki and Gries suggest introducing auxiliary (typically unbounded, "history") variables, while Lamport suggests making

some local state globally visible. In essence, both methods expose information about the internal behavior of the component processes.

History variables, which are often valuable for a deductive proof, complicate model checking, as the generic construction turns a finite-state problem into an infinite-state one. Lamport's proposal retains the finite-state nature of the problem, but it is not clear how to choose the local state to be exposed. (Exposing *all* local state results in a global model checking problem.) A second contribution of this work is a systematic refinement scheme, which is based on an analysis of counter-examples produced for the abstracted processes.

This combination of local reasoning with refinement is sound for all programs, and is shown to be complete for finite-state programs. I.e., for a finite-state program, a property is eventually proved or disproved. The combined method is simple to implement in terms of BDD manipulations. Experiments with an implementation based on TLV [33] show that local reasoning can have a significant advantage over global reasoning. Full proofs and more experiments are in [10].

2 Motivating Example



a: protocol MUX-SEM   b: abstracted form of process $P[i]$

Fig. 1. Example Illustrating the Local Reasoning Method.

Figure 1(a) shows a very simple mutual exclusion protocol. It satisfies mutual exclusion (a safety property), and the following (weak) progress property: "infinitely often $x = 0$", i.e., infinitely often, some process is in its critical section. It does not satisfy the stronger individual progress property "every waiting process eventually enters its critical section". In the following, consider a 2-process instance.

The first step of the local method is to compute the strongest *split invariant*, θ, which is the strongest vector of local, per-process inductive invariants. (The reachable states form the strongest *global* inductive invariant.) This computation, by the fixpoint method in [30], results in $\theta_i = \mathit{true}$, for all i.

The second step uses θ to compute an over-approximation $\overline{P[\theta, i]}$ of each process P_i. For process i, the transition relation of $\overline{P[\theta, i]}$ is the disjoint union of (a) the transition relation, T_i, of P_i and (b) for every other process, a transition relation which summarizes its effect on the *shared state*, under the constraint θ. The contribution for process P_j ($j \neq i$) is computed by quantifying out the local

variables of process P_j (both current and next-state) from the formula $\theta_j \wedge T_j$. For this protocol, the summary for P_j allows x to change arbitrarily, leading to the abstract process shown in Figure 1(b).

The final step is to check, using a standard model checker whether, for *all* i, the abstraction $\overline{P[\theta, i]}$ satisfies a weakening of the original progress property: that there is no execution on which a Büchi automaton for the negated original property accepts infinitely often from T_i transitions. This check succeeds for both abstract processes, which proves the original property for the original program.

It is interesting to note that the abstract form of process P_1 does fail the *original* property, as the summary transition for process P_2 can ensure that x is "stuck" at 1 forever. However, in this execution, a T_1 transition does not appear infinitely often, so it does not represent a failure for the weaker property defined above.

The particular form of these checks arises naturally from the derivation that follows. In broad brush, however, the steps can be seen as: (1) constraining the globally reachable states with a split invariant, (2) computing an over-approximation of the behavior of each component process, restricted by the split invariant and (3) checking a weakened property on these over-approximations. In each step, the method avoids computing with the global product state space.

Although it is not the case for this protocol, it is possible that the abstraction of step (2) is too weak, resulting in false error reports in step (3). Exposing more local information strengthens the split invariant at the next iteration and, consequently, tightens the process summaries computed in step (2). This eventually results in the elimination of false errors.

Finally, for this protocol, the local proof for the 2-process instance suffices to show that the property holds for *all* instances (i.e., in a parameterized sense). This is true as both the split invariant $\theta = \mathit{true}$, and the abstract process $\overline{P[\theta, 1]}$ are unchanged for larger instances, and $\overline{P[\theta, j]}$ is identical to $\overline{P[\theta, 1]}$ by symmetry. We do not explore such parameterized proofs further in this paper.

3 The Local Reasoning Method

This section defines the computational model, and presents the derivation of the local reasoning method. Some of the preliminary definitions are taken from [30], and are repeated here for convenience.

3.1 Basic Definitions

Programs and Composition A *program* is defined as 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.

Programs need not be finite-state. The local reasoning method is sound in general, but we show completeness only for finite-state programs.

The semantics of a program is given by a *transition system*, which is defined by 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* (or *assertion*) is a Boolean expression over the program variables. The value of a variable w at state s is denoted by $w(s)$. The truth value of a predicate p at a state s, denoted by $p(s)$, is defined as usual by induction on formula structure.

Given a non-empty, finite, set of programs, $\{P_i\}$, their *asynchronous composition*, written as $(||i :: P_i)$, is the program $P = (V, I, T)$, with components defined as follows. Let $V = (\bigcup i :: V_i)$ and $I = (\bigwedge i :: I_i)$. The *shared variables*, denoted X, are those that belong to $V_i \cap V_j$, for a distinct pair (i, j). For simplicity, we assume that $X \subseteq V_i$, for all i. The *local variables* of P_i, denoted L_i, are the non-shared variables in V_i (i.e., $L_i = V_i \setminus X$). We assume that the local variables of distinct processes are disjoint. The set of local variables is $L = (\bigcup i :: L_i)$. The transition condition T_i of program P_i is constrained to leave local variables of other processes unchanged. Define $\hat{T}_i$ as $T_i(V_i, V_i') \wedge unch(L \setminus L_i)$, where $unch(W)$, for a set of variables W, is defined as $(\forall w : w \in W : w' = w)$. Then T is defined simply as $(\bigvee i :: \hat{T}_i)$.

Inductiveness and Invariance A state predicate φ is an *invariant* of program $P = (V, I, T)$ if it holds at all reachable states of P. A state assertion ξ is an *inductive invariant* for P if it is initial (1) and inductive (2).

$$[I \Rightarrow \xi] \tag{1}$$

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

Notation: *wlp* is the *weakest liberal precondition* transformer introduced by Dijkstra in [14]. The notation $[\psi]$, from [15], is read as "ψ is valid". For $P = (||i :: P_i)$, it is the case that $[wlp(T, \varphi) \equiv (\bigwedge i :: wlp(\hat{T}_i, \varphi))]$.

Split Invariance A *local assertion*, θ_i, is defined over the variables of P_i. Thus, θ_i is defined over L_i and X, but *does not* refer to other local variables. A *split assertion* for a composition $(||i :: P_i)$ is a vector of local assertions, one for each process. A split assertion, θ, is a *split invariant* if the conjunction $(\wedge i :: \theta_i)$ is an inductive invariant for the composition. To simplify notation, θ refers indifferently either to the vector or to the conjunction of its components, with the interpretation clear from the context. In [30], it is shown that the *strongest* split invariant can be computed as the (simultaneous) least fixpoint of the set of equations below, one for each i.

$$[\theta_i \equiv (\exists L \setminus L_i :: I \vee (\vee j :: sp(\hat{T}_j, \theta)))]$$

Here, *sp* is the strongest post-condition operator (also known as "post"). The expression takes successors of θ (i.e., of $(\wedge i :: \theta_i)$) for each $\hat{T}_j$, adds the initial states, and quantifies out non-P_i local variables, to ensure that the result is a local assertion for P_i. For finite-state programs, these calculations are easily implemented with standard BDD operations.

3.2 Background: Proofs of Linear-time Properties

The automata-theoretic approach to model checking [37] is followed here. Linear-time properties are specified by a finite-state automaton over infinite words for the *complemented* property. The derivations below rely on a few assumptions.

1. *Programs are deadlock-free.* As the transition relation of each component is left-total, the program as a whole never gets stuck. As in the UNITY model [3], however, "deadlock" may be defined as a state where the only transition is a self-loop. Deadlock-freedom is thus a safety property, which can be checked locally using the method from [9].
2. *The property is defined by a non-deterministic Büchi automaton for its complement, and refers only to the shared variables.* This enables the automaton transitions to be inserted into the program, synchronously with each component transition. The predicate *accept*, on the shared state, indicates that the automaton is in an accepting state.
3. *Fairness constraints are enforced by the automaton.* Liveness properties often depend on fairness assumptions about execution schedules. For simplicity, we assume that any fairness conditions are part of the automaton; i.e., the automaton accepts an execution if it is fair but fails the property.

Under these assumptions, the following non-compositional proof rule can be used to prove a linear-time property for a program $P = (V, I, T)$. The proof rule requires an assertion θ, and a *rank function* ρ, a partial map from states to a well-founded domain, $(W, \prec)$, which satisfy the conditions below.

$$[I \;\Rightarrow\; \theta] \tag{3}$$

$$[\theta \;\Rightarrow\; wlp(T, \theta)] \tag{4}$$

$$[\theta \;\Rightarrow\; domain(\rho)] \tag{5}$$

$$\forall k : k \in W : [\theta \wedge (\rho = k) \;\Rightarrow\; wlp(T, \rho \preceq k)] \tag{6}$$

$$\forall k : k \in W : [\theta \wedge accept \wedge (\rho = k) \;\Rightarrow\; wlp(T, \rho \prec k)] \tag{7}$$

Theorem 0 *The proof rule is sound and relatively complete.*

3.3 Localizing the Proof Rule

Consider now the case where P is a composition $(||i : P_i)$. The goal is to *localize* the reasoning rule given previously. To this end, a first change is to make θ a *split invariant*. A second change is to let ρ be a *vector* of local functions, with ρ_i defined over the variables V_i of P_i, with a well-founded domain, $(W_i, \prec_i)$, as

its range. The local proof rule is as follows.

$$[I \Rightarrow \theta] \tag{8}$$

$$[\theta \Rightarrow wlp(T, \theta)] \tag{9}$$

$$\forall i : [\theta \Rightarrow domain(\rho_i)] \tag{10}$$

$$\forall i, k : k \in W_i : [\theta \wedge (\rho_i = k) \Rightarrow wlp(T, \rho_i \preceq_i k)] \tag{11}$$

$$\forall i, k : k \in W_i : [\theta \wedge accept \wedge (\rho_i = k) \Rightarrow wlp(\hat{T}_i, \rho_i \prec_i k)] \tag{12}$$

Theorem 1 *The local proof rule is sound.*
Proof Sketch. The first two conditions ensure that θ is a (split) inductive invariant. Define a global rank function ρ by $\rho(s) = (vec\ i :: \rho_i(s_i))$, where s_i is s restricted to V_i. Global rank vectors are compared point-wise. From (10)-(12), it follows that the pair (θ, ρ) satisfies the hypotheses of the previous proof rule, ensuring soundness by Theorem 0. □

Condition (11) ensures that ρ_i is not adversely affected by any transition of P. This is one of the "non-interference" properties defined by Owicki and Gries in [31]. The local formulation has the following interesting consequence.

Theorem 2 *If the local proof rule is applicable, it can be applied with θ being the* strongest *split invariant.*
Proof. Suppose that the local proof rule is applicable for some θ. Let θ^* represent the strongest split invariant. Conditions (8) and (9) are satisfied by θ^* by definition. The other conditions are anti-monotone in θ; as θ^* is stronger than θ, they hold also for θ^*. □

This theorem provides the first hint for mechanizing the local proof rule, as it eliminates one part of the guesswork: one can let θ be the strongest split invariant. The next section shows how to replace the rank function requirements with model checking.

3.4 Guessing Ranks through Model Checking

In this section, we consider a *fixed* split invariant, θ. The goal is to replace the reasoning about rank functions with a local model checking procedure. First, note that by the conjunctivity of *wlp* for asynchronous composition, conditions (11) and (12) are equivalent to saying that, for each i, each j, and $k \in W_i$,

$$[\theta \wedge (\rho_i = k) \Rightarrow wlp(\hat{T}_j, \rho_i \preceq_i k)] \tag{13}$$

$$[\theta \wedge accept \wedge (\rho_i = k) \Rightarrow wlp(\hat{T}_i, \rho_i \prec_i k)] \tag{14}$$

These conditions are rewritten below, exploiting locality. In these derivations, we do not explicitly write the variable dependencies, to avoid clutter. For reference, they are: $\theta(X, L)$, $\theta_i(X, L_i)$, $\rho_i(X, L_i)$, $\hat{T}_i(X, L, X', L')$, and $T_i(X, L_i, X', L'_i)$. We write ρ'_i to refer to $\rho_i(X', L'_i)$. The calculations make extensive use of the following fact: $[p(x, y) \Rightarrow q(y)]$ is equivalent to $[(\exists x :: p(x, y)) \Rightarrow q(y)]$.

For each i, each j, and $k \in W_i$,

$$
\begin{array}{ll}
 & [\theta \wedge (\rho_i = k) \;\Rightarrow\; wlp(\hat{T}_j, \rho_i \preceq_i k)] \\
\equiv & \quad \{ \text{ definition of } wlp \text{ } \} \\
 & [\theta \wedge (\rho_i = k) \wedge \hat{T}_j \;\Rightarrow\; \rho_i' \preceq_i k] \\
\equiv & \quad \{ \text{ by locality, as the consequent is independent of } L \setminus L_i \text{ and } L' \setminus L_i' \text{ } \} \\
 & [(\exists L \setminus L_i, L' \setminus L_i' :: \theta \wedge (\rho_i = k) \wedge \hat{T}_j) \;\Rightarrow\; \rho_i' \preceq_i k] \\
\equiv & \quad \{ \text{ pushing quantifiers inwards } \} \\
 & [(\exists L \setminus L_i :: \theta \wedge (\exists L' \setminus L_i' :: \hat{T}_j)) \wedge (\rho_i = k) \;\Rightarrow\; \rho_i' \preceq_i k]
\end{array}
$$

For $j \neq i$, the term $(\exists L' \setminus L_i' :: \hat{T}_j)$ simplifies to $(\exists L_j' :: T_j) \wedge unch(L_i)$. As θ is really $(\wedge i :: \theta_i)$, the final implication simplifies to

$$
[(\exists L \setminus L_i :: \theta) \wedge (\exists L_j : \theta_j \wedge (\exists L_j' :: T_j)) \wedge unch(L_i) \wedge (\rho_i = k) \;\Rightarrow\; \rho_i' \preceq_i k] \quad (15)
$$

This has a shape similar to that of the condition (6) from the non-local rule, with $(\exists L \setminus L_i :: \theta)$ playing the role of the invariant assertion, and the term "$(\exists L_j : \theta_j \wedge (\exists L_j' :: T_j)) \wedge unch(L_i)$" playing the role of a transition relation. This observation leads to the following definition.

Definition 0 *For j distinct from i, define $\overline{T_j[\theta, i]}$ as $(\exists L_j :: \theta_j \wedge (\exists L_j' :: T_j)) \wedge unch(L_i)$. With free variables (X, L_i, X', L_i'), this is a transition term for P_i.*

Similarly transforming the other conditions, one obtains for any i, and any $j : j \neq i$,

$$
\begin{array}{lr}
[(\exists L \setminus L_i :: init) \;\Rightarrow\; (\exists L \setminus L_i :: \theta)] & (16) \\
[(\exists L \setminus L_i :: \theta) \;\Rightarrow\; wlp(T_i, (\exists L \setminus L_i :: \theta))] & (17) \\
[(\exists L \setminus L_i :: \theta) \;\Rightarrow\; wlp(\overline{T_j[\theta, i]}, (\exists L \setminus L_i :: \theta))] & (18) \\
[(\exists L \setminus L_i :: \theta) \;\Rightarrow\; domain(\rho_i)] & (19) \\
\forall k : k \in W_i : [(\exists L \setminus L_i :: \theta) \wedge (\rho_i = k) \;\Rightarrow\; wlp(\overline{T_j[\theta, i]}, \rho_i \preceq_i k)] & (20) \\
\forall k : k \in W_i : [(\exists L \setminus L_i :: \theta) \wedge accept \wedge (\rho_i = k) \;\Rightarrow\; wlp(T_i, \rho_i \prec_i k)] & (21)
\end{array}
$$

The implications (16)-(21) suggests the definition of an abstract process.

Definition 1 *The abstraction of process P_i relative to θ is a process denoted $\overline{P[\theta, i]}$, with variables V_i, initial condition $(\exists L \setminus L_i :: I_i)$, and transition relation formed by the terms T_i and $\overline{T_j[\theta, i]}$, for $j : j \neq i$, combined disjunctively.*

Conditions (20) and (21) lead to the following theorem.

Theorem 3 *For fixed θ: if there is a rank function vector which satisfies the local proof conditions then, for any i, $\overline{P[\theta, i]}$ satisfies the property "for all executions, it is not the case that T_i occurs infinitely often from states satisfying accept".*

The contrapositive of this theorem implies that, for a given θ, if the check fails for one of the abstract processes, there is *no* rank function vector which can satisfy the local proof rule (for the same θ). This forces a refinement of the split invariant in order to rule out false errors, as described in the next section. On the other hand, as shown below, if the check succeeds for *all* of the abstract processes, the property must hold of the original program.

Theorem 4 *For any split invariant θ: if, for every i, $\overline{P[\theta, i]}$ satisfies the property "for all executions, it is not the case that T_i occurs infinitely often from states satisfying accept", then the original property is true of the composition $(||i :: P_i)$.*

3.5 The Local Reasoning Algorithm

The algorithm is now easily stated. Given $P = (||i :: P_i)$, with an embedded Büchi automaton for a negated property, and acceptance condition *accept*.

1. Compute a split invariant θ of P, ideally the strongest split invariant.
2. For each i, define the abstract program $\overline{P[\theta, i]}$ (Def. 1). Form the product of the abstract program with the property automaton. Check the property stated in Theorem 3.
3. If each check succeeds, by Theorem 4, the property holds of P.

Computing the split invariant, and checking the property, are operations that are polynomial in the number of processes and the size of each process and the automaton.

What if the check fails for some i? By Theorem 2, if the strongest split invariant was used, it is either the case that the property is false, or that a refinement step is needed to expose more of the local state and rule out a false counter-example.

3.6 Modifications

Quantified Properties For parameterized protocols, it is common to have quantified liveness properties (e.g., "every waiting process eventually enters its critical section"). Fortunately, such protocols typically have a high degree of symmetry. Under symmetry, a composition $(||i :: P_i)$ satisfies a quantified property $(\forall i :: \varphi(i))$ if, and only if, it satisfies $\varphi(1)$ [16]. Hence, making the local variables of P_1 part of the shared state suffices to meet the requirement that the property is defined over the shared variables. Symmetry can also be exploited to reduce the computations for the split invariance calculation, and to reduce the number of checks needed in step 2 above to the abstract processes for P_1 and P_2.

Fairness For an unconditional fairness assumption, it suffices to annotate each transition with the index of the process making the transition. These indices are carried over to the abstract processes. To express stronger fairness assumptions, it is necessary to shadow the local predicates mentioned in the fairness assumption with auxiliary shared variables.

4 A Refinement Strategy

The local reasoning algorithm defined above is necessarily incomplete. If the property cannot be proved, a local proof requires exposing more of the local state. We describe a simple, yet effective, strategy to choose the portions of the local state to be exposed. This strategy is based on examining counter-example executions for those abstract processes which fail to model-check.

A failure for $\overline{P[\theta, i]}$ implies that there is a finite, "lasso" shaped counter-example: a path ending in a cycle which contains at least one T_i transition from an *accept* state. By construction, in $\overline{P[\theta, i]}$, the T_i transitions are exact, while the transitions of other processes may be approximate. Recall that the definition of $\overline{T_j[\theta, i]}$ (Def. 0) has an $\exists L_j \exists L'_j$ form. In terms of language used in branching-time abstraction methods [12], this is a *may*-transition. It is a *must*-transition if, for *every* value of L_j that satisfies θ_j, there is a T_j transition from (X, L_j) to (X', L_j). The distinction between may- and must-transitions is useful in determining whether a counterexample lasso represents a real execution.

Theorem 5 *Let π be a counterexample for $\overline{P[\theta, i]}$. If every abstract transition along π is also a must-transition, there is an induced global counterexample for the full program P.*

This theorem leads to the refinement procedure below.

1. If, for some abstract program $\overline{P[\theta, i]}$, the transitions on a counterexample path, π, meet the condition of Theorem 5, HALT("the property is false"), and provide the induced global path as a counterexample.
2. Otherwise, let $t = (u, \overline{T_k[\theta, i]}, v)$ be a non-must transition on π. Define p_k by

$$p_k(L_k) := \theta_k(X(u), L_k) \wedge \neg(\exists L'_k :: T_k(X(u), L_k, X(v), L'_k))$$

   For local state a of P_k, $p_k(a)$ is true if a is an *obstacle* to forming a must-transition, since there is no transition to $X(v)$ from $(X(u), a)$ in P_k. Predicate p_k is non-trivial: it is not valid, as t is a may transition; neither is it unsatisfiable, as t is not a must transition. As required for local reasoning, p_k is a local assertion for P_k. A *shared* boolean variable, b_k, is added to the program, such that $b_k \equiv p_k(L_k)$ is an invariant, and the local reasoning algorithm is repeated for the augmented program. The initial value of b_k is the initial value of p_k; the constraint $(b'_k \equiv p_k(L'_k))$ is conjoined to T_k, and $(b'_k \equiv b_k)$ to T_j, for $j \neq k$.

Theorem 6 *For finite-state programs, this procedure eventually terminates.*

Proof. First, we show that each refinement step discovers at least one new predicate. Existing predicates are preserved by the split invariance calculation (Lemma 1 of [9]). Thus, any existing predicates have the same value for all local states a of P_k that satisfy $\theta_k(X(u), a)$; and this is not true of p_k by its definition. As there are a bounded number of predicates, the refinement process cannot continue forever. $\square$

Theorem 7 *The combination of local reasoning with refinement is complete for finite-state programs.*

Proof. Consider first the case where the property holds. Thus, any counterexamples are not real, so the hypothesis of Theorem 5 does not apply, and the procedure will not terminate with an incorrect answer. As termination is guaranteed by Theorem 6, the procedure must terminate with success.

Next, consider the case where the property does not hold. By the contrapositive of Theorem 4, at every stage, at least one of the abstract processes fails the check. Thus, the procedure cannot terminate with success. As termination is guaranteed by Theorem 6, the procedure must terminate with failure. □

While the termination argument relies on exhausting the set of available predicates, the hope is that, in most cases, termination occurs before the problem is transformed back into a global model checking question. It is important to note that the procedure is *sound*—but not necessarily terminating—for all programs.

5 Experiments

We implemented our method in TLV [33], a BDD-based model checker. The experiments use parameterized protocols, as the global state space can be varied simply by altering the number of processes. We do not use symmetry to optimize the calculations, as the intent is to compare local with global reasoning, as represented by algorithm TEMP_ENTAIL, based on [27]. The experiments show that local reasoning can have significantly better performance than global reasoning.

Method	Processes	BDDs	Time (sec)	BDDs [variant]	Time (sec) [variant]
Local Reasoning	2	433	0	7.6k	0
Global Reasoning	2	440	0	6.2k	0
Local Reasoning	10	10k	0.1	19k	1.3
Global Reasoning	10	10k	0.05	248k	294
Local Reasoning	20	15k	1.28	62k	4.6
Global Reasoning	20	23k	1.53	-	>2hrs
Local Reasoning	50	88k	21.7	330k	46.8
Global Reasoning	50	141k	53.5	-	>2hrs

Table 1. Test results for the property $\Box \Diamond x = 0$.

We checked two different properties for MUX-SEM, the motivating example of Figure 1(a). For the first property, $\Box \Diamond x = 0$ ("infinitely often $x = 0$"), our method does not require any refinement step. Compared to TEMP_ENTAIL, it runs significantly faster and requires nearly half the amount of BDDs (Table 1). A variant of the protocol models the situation where there is some state irrelevant to the property (in this case, a counter in each process). The locality of the analysis results in this excess state being eliminated from the abstract processes. The effect is shown in the two final columns of Table 1.

The second property is $\Box(P[1].at_l_1 \rightarrow \Diamond P[1].at_l_2)$ ("if process $P[1]$ is at location 1 it eventually enters its critical section"), which is not satisfied by MUX-SEM under unconditional fairness. Expressing the property requires exposing the location variable of $P[1]$. The method detects a counter example after one refinement step, during which one bit of information per process (whether the process is waiting) is exposed. The results are provided in Table 2.

Method	Processes	BDDs	Time (sec)	Refinements	New Variables
Local Reasoning	3	2.6k	0.02	1	1
Global Reasoning	3	1.6k	0.01	-	-
Local Reasoning	10	10k	0.22	1	8
Global Reasoning	10	16k	0.13	-	-
Local Reasoning	20	37k	1.2	1	18
Global Reasoning	20	66k	1.46	-	-
Local Reasoning	50	215k	13.5	1	48
Global Reasoning	50	414k	30.6	-	-
Local Reasoning	100	852k	382	1	98
Global Reasoning	100	1.6M	586	-	-

Table 2. Test results for the property $\Box(P[1].at_l_1 \rightarrow \Diamond P[1].at_l_2)$.

6 Related Work and Conclusions

Compositional reasoning about concurrency has a long history, going back 30 years to the seminal papers of Owicki and Gries, and Lamport. Early work focuses on deductive proof methods for safety [4, 26] and liveness [32, 13]. Tools such as Cadence SMV support guided compositional proofs [29, 25]. "Thread-modular" reasoning[18, 19, 24, 23] uses per-process transition relations to prove safety, but the method is incomplete. The split invariance method was introduced in [30], with completeness obtained by a refinement method [9].

The new contribution here is the mechanization of an Owicki-Gries style proof rule for liveness properties, coupled with a refinement procedure. The procedure is fully automatic, and complete for finite-state processes. It has a simple implementation, and the experimental results support the hypothesis that local reasoning is often significantly faster than global reasoning. To the best of our knowledge, this is the first fully automated and complete method of its type for general linear-time temporal properties of asynchronous programs.

Recent work [22, 11] has shown that local reasoning can be effective for proving termination properties. However, the algorithms do not include a mechanism to expose additional local state, which is necessary for completeness.

In [35], Shoham and Grumberg propose a complete compositional method, coupled with refinement, for proving mu-calculus properties. Our methods have several points of commonality, including the use of the may-must distinction for refinement, but also some important differences: the method in [35] operates on

synchronous compositions, rather than the asynchronous compositions considered here, and has as a key step the global analysis of a composition of abstract processes, which differs from the separate analysis of individual abstract processes in our method. Earlier work [1] uses per-process invariants to constrain abstractions (in the synchronous setting) as is done in Definitions 0 and 1.

Automata learning algorithms have been used for compositional analysis of safety properties [21, 36, 20, 2], and recently extended to liveness properties [17]. The algorithms are complete, but can be expensive in practice [8]. Our completion procedure is based on a form of counter-example guided refinement [7], which may be viewed as a process of learning from failure.

Acknowledgements This research was supported in part by the National Science Foundation under grant CCR-0341658.

References

1. R. Alur, L. de Alfaro, T. A. Henzinger, and F. Y. C. Mang. Automating modular verification. In *CONCUR*, volume 1664 of *LNCS*, pages 82–97, 1999.
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. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.
4. K.M. Chandy and J. Misra. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, 7(4), 1981.
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. E. M. Clarke and O. Grumberg. Avoiding the state explosion problem in temporal logic model checking. In *PODC*, pages 294–303, 1987.
7. 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.
8. 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.
9. A. Cohen and K. S. Namjoshi. Local proofs for global safety properties. In *CAV*, volume 4590 of *LNCS*, pages 55–67. Springer, 2007.
10. A. Cohen and K. S. Namjoshi. Local proofs for linear-time temporal properties of concurrent programs. Technical report, Bell Labs, 2008. Available at `http://www.cs.bell-labs.com/who/kedar`.
11. B. Cook, A. Podelski, and A. Rybalchenko. Proving thread termination. In *PLDI*, pages 320–330. ACM, 2007.
12. D. Dams, O. Grumberg, and R. Gerth. Abstract interpretation of reactive systems. *TOPLAS*, 19(2), 1997.
13. 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.

14. E.W. Dijkstra. Guarded commands, nondeterminacy, and formal derivation of programs. *CACM*, 18(8), 1975.
15. E.W. Dijkstra and C.S. Scholten. *Predicate Calculus and Program Semantics.* Springer Verlag, 1990.
16. E.A. Emerson and A.P. Sistla. Symmetry and model checking. In *CAV*, volume 697 of *LNCS*, 1993.
17. A. Farzan, Y. Chen, E. M. Clarke, Y. Tsan, and B. Wang. Extending automated compositional verification to the full class of omega-regular languages. In *TACAS*, LNCS, 2008.
18. 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.
19. C. Flanagan and S. Qadeer. Thread-modular model checking. In *SPIN*, volume 2648 of *LNCS*, pages 213–224, 2003.
20. D. Giannakopoulou and C. S. Pasareanu. Learning-based assume-guarantee verification (tool paper). In *SPIN*, volume 3639 of *LNCS*, pages 282–287, 2005.
21. D. Giannakopoulou, C. S. Pasareanu, and H. Barringer. Assumption generation for software component verification. In *ASE*, pages 3–12, 2002.
22. A. Gotsman, J. Berdine, B. Cook, and M. Sagiv. Thread-modular shape analysis. In *PLDI*, pages 266–277. ACM, 2007.
23. T. A. Henzinger, R. Jhala, and R. Majumdar. Race checking by context inference. In *PLDI*, pages 1–13, 2004.
24. T. A. Henzinger, R. Jhala, R. Majumdar, and S. Qadeer. Thread-modular abstraction refinement. In *CAV*, volume 2725 of *LNCS*, pages 262–274, 2003.
25. R. Jhala and K. L. McMillan. Microarchitecture verification by compositional model checking. In *CAV*, volume 2102 of *LNCS*, pages 396–410. Springer, 2001.
26. C.B. Jones. *Development methods for computer programs including a notion of interference*. PhD thesis, Oxford University, 1981.
27. Y. Kesten, A. Pnueli, L. Raviv, and E. Shahar. Model checking with strong fairness. *Formal Methods in System Design*, 28(1):57–84, 2006.
28. L. Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. Software Eng.*, 3(2), 1977.
29. K.L. McMillan. A compositional rule for hardware design refinement. In *CAV*, volume 1254 of *LNCS*, 1997.
30. K. S. Namjoshi. Symmetry and completeness in the analysis of parameterized systems. In *VMCAI*, volume 4349 of *LNCS*, 2007.
31. S. S. Owicki and D. Gries. Verifying properties of parallel programs: An axiomatic approach. *Commun. ACM*, 19(5):279–285, 1976.
32. A. Pnueli. In transition from global to modular reasoning about programs. In *Logics and Models of Concurrent Systems*, NATO ASI Series, 1985.
33. 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`.
34. 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.
35. S. Shoham and O. Grumberg. Compositional verification and 3-valued abstractions join forces. In *SAS*, volume 4634 of *LNCS*, pages 69–86, 2007.
36. O. Tkachuk, M. B. Dwyer, and C. S. Pasareanu. Automated environment generation for software model checking. In *ASE*, pages 116–129, 2003.
37. M. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *IEEE Symposium on Logic in Computer Science*, 1986.