Commutativity For Concurrent Program Termination Proofs

Danya Lette[0000−0002−3929−7521] and Azadeh Farzan[0000−0001−9005−2653]

University of Toronto, Toronto, Canada
{danya,azadeh}@cs.toronto.edu

Abstract. This paper explores how using commutativity can improve the efficiency and efficacy of algorithmic termination checking for concurrent programs. If a program run is terminating, one can conclude that all other runs equivalent to it up-to-commutativity are also terminating. Since reasoning about termination involves reasoning about infinite behaviours of the program, the equivalence class for a program run may include infinite words with lengths strictly larger than ω that capture the intuitive notion that some actions may soundly be postponed indefinitely. We propose a sound proof rule which exploits these as well as classic bounded commutativity in reasoning about termination, and devise a way of algorithmically implementing this sound proof rule. We present experimental results that demonstrate the effectiveness of this method in improving automated termination checking for concurrent programs.

1 Introduction

Checking termination of concurrent programs is an important practical problem and has received a lot of attention [37,35,29,3]. A variety of interesting techniques, including thread-modular reasoning [34,10,37,35], causality-based reasoning [29], and well-founded proof spaces [15], among others, have been used to advance the state of the art in reasoning about concurrent program termination. Independently, it has been established that leveraging *commutativity* in proving safety properties can be a powerful tool in improving automated checkers [18,19,16,17]. There are many instances of applications of Lipton's reductions [32] in program reasoning [14,28]. Commutativity has been used to simultaneously search for a program with a simple proof and its safety proof [18,19] and to improve the efficiency and efficacy of assertion checking for concurrent programs [16]. Recently [17], *abstract commutativity* relations are formalized and combined to increase the power of commutativity in algorithmic verification.

This paper investigates how using commutativity can improve the efficiency and efficacy of proving the termination of concurrent programs as an enhancement to existing techniques. The core idea is simple: if we know that a program run $\rho ab\rho'$ is terminating, and we know that a and b commute, then we can conclude that $\rho ba\rho'$ is also terminating. Let us use an example to make this idea concrete for termination proofs of concurrent programs. Consider the two thread templates in Figure 1: one for a producer thread and one for a consumer thread,

Producer Thread:

```
while (i < producer_limit){
   C++; // produce content
   i++
}
barrier++;
```

Consumer Thread:

```
assume(barrier >= producer_num);
while (j < consumer_limit){
   j--;
   C--; // consume content
}
```

Fig. 1: Producer/Consumer Template

where `i` and `j` are local variables. The assumption is that `barrier` and the local counters `i` and `j` are initialized to 0. The producer generates content (modelled by incrementing of a global counter `C++`) up to a limit and then, using `barrier`, signals the consumer to start consuming. Independent of the number of producers and consumers, this synchronization mechanism ensures that the consumers wait for all producers to finish before they start consuming. Note that the producer threads fully commute — each statement in a producer commutes with each statement in another. A producer and consumer only partially commute.

In a program with only two producers, a human would argue at the high level that the *independence* of producer loops implies that their parallel composition is equivalent, up to *commutativity*, to their sequential composition. Therefore, it suffices to prove that the sequential program terminates. In other words, it should suffice to prove that each producer terminates. Let us see how this high level argument can be formalized using commutativity reasoning. Let λ_1 and λ_2 stand for the loop bodies of the two producers. Among others, consider the (syntactic) concurrent program run $(\lambda_1\lambda_2)^\omega$; this run may or may not be feasible. Since λ_1 and λ_2 commute, we can transform this run, by making *infinitely many* swaps, to the run $\lambda_1^\omega\lambda_2^\omega$. The model checking expert would consider this transformation rather misguided: it appears that we are indefinitely postponing λ_2 in favour of λ_1. Moreover, a word with a length strictly larger than ω, called a *transfinite* word, does not have an appropriate representation in language theory because it does not belong to Σ^ω. Yet, the observation that $(\lambda_1\lambda_2)^\omega \equiv \lambda_1^\omega\lambda_2^\omega$ is the key to a powerful proof rule for termination of concurrent programs: If λ_1^ω is terminating and λ_1 commutes against λ_2, then we can conclude that $(\lambda_1\lambda_2)^\omega$ is terminating. In other words, the termination proof for the first producer loop implies that all interleaved executions of two producers terminate, without the need for a new proof. Note that the converse is not true; termination of $\lambda_1^\omega\lambda_2^\omega$ does not necessarily imply the termination of λ_2^ω. So, even if we were to replace the second producer with a forever loop, our observation would stand as is. Hence, for the termination of the entire program (and not just the run $(\lambda_1\lambda_2)^\omega$), one needs to argue about the termination of both λ_1^ω and λ_2^ω, matching the high level argument. In Section 3, we formally state and prove this proof rule, called the *omega-prefix* proof rule, and show how it can be incorporated into an algorithmic verification framework. Using this proof rule, the program consisting of N producers can be proved terminating by proving precisely N single-thread loops terminating.

Now, consider adding a consumer thread to our two producer threads. The consumer loop is independent of the producer threads but the consumer thread,

as a whole, is not. In fact, part of the work of a termination prover is to prove that any interleaved execution of a consumer loop with either producer is *infeasible* due to the *barrier* synchronization and therefore terminating. Again, a human would argue that two such cases need to be considered: the consumer crosses the barrier with 0 or 1 producers having terminated. Each case involves several interleavings, but one should not have to prove them correct individually. Ideally, we want a mechanism that can take advantage of commutativity for both cases.

Before we explore this further, let us recall an algorithmic verification template which has proven useful in incorporating commutativity into safety reasoning [18,19,16,17] and in proving termination of sequential [25] and parameterized concurrent programs [15]. The work flow is illustrated in Figure 2. The program and the proof are represented using (Büchi) automata, and module (d) (and consequently module (a)) are implemented as inclusion checks between the languages of these automata. The iteratively refined proof — a language of infeasible syntactic program runs — can be annotated Floyd-Hoare style and generalized using interpolation as in [25]. For module (b), any known technique for reasoning about the termination of simple sequential programs can be used on lassos.

The straightforward way to account for commutativity in this refinement loop would involve module (c): add to Π all program runs equivalent to the existing ones up to commutativity without having a proof for them. In the safety context, it is well-known that checking whether a program is subsumed by the *commutativity closure* of a proof is *undecidable*. We show (in Section 3) that the same hurdle exists when doing inclusion checks for program termination.

In the context of safety [18,19,16,17], *program reductions* were proposed as an antidote to this undecidability problem: rather than enlarging the proof, one reduces the program and verifies a new program with a subset of the original program runs while maintaining (at least) one representative for each commutativity equivalence class. These representatives are the lexicographically least members of their equivalence classes, and are algorithmically computed based on the idea of the *sleep set* algorithm [22] to construct the automaton for the reduced program. However, using this technique is not possible in termination reasoning where *lassos*, and not finite program runs, are the basic objects.

To overcome this problem, we propose a different class of reductions, called *finite-word reduction*. Inspired by the classical result that an ω-regular language can be faithfully captured as a finite-word language for the purposes of certain checks such as inclusion checks [4], we propose a novel way of translating both the program and the proof into finite-word languages. The classical result is based on

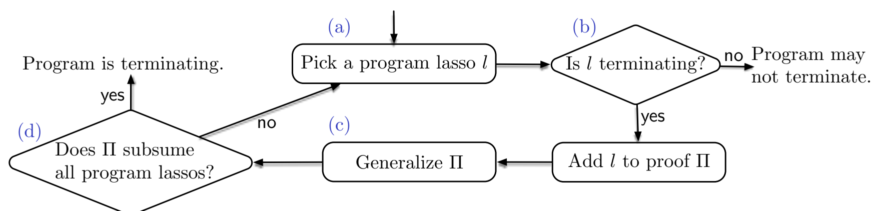


Fig. 2: Refinement Loop For Proving Termination.

an exponentially sized construction and does not scale. We propose a polynomial construction that has the same properties for the purpose of our refinement loop. This contribution can be viewed as an efficient translation of termination analysis to safety analysis and is useful independent of the commutativity context. For the resulting finite-word languages, we propose a novel variation of the *persistent set* algorithm to *reduce* the finite-word program language. This reduction technique is aware of the lasso structure in finite words.

Used together, finite-word reductions and omega-prefix generalization provide an approximation of the undecidable commutativity-closure idea discussed above. They combine the idea of closures, from proof generalization schemes like [15] and reductions from safety [16], into one uniform proof rule that both *reduces* the program and *generalizes* the proof up to commutativity to take as much advantage as possible. Neither the reductions nor the generalizations are ideal, which is a necessity to maintain algorithmic computability. Yet, together, they can perform in a near optimal way in practice: for example, with 2 producers and one consumer, the program is proved terminating by sampling precisely 3 terminating lassos (1 for each thread) and 2 infeasible lassos (one for each barrier failure scenario).

Finally, mostly out of theoretical interest, we explore a class of infinite word reductions that have the same theoretical properties as safety reductions, that is, they are *optimal* and their regularity (in this case, ω-regularity) is guaranteed. We demonstrate that if one opts for the *Foata Normal Form (FNF)* instead of lexicographical normal form, one can construct optimal program reductions in the style of [18,19,16] for termination checking. To achieve this, we use the notion of the FNF of infinite words from (infinite) trace theory [13], and prove the ω-regular analogue of the classical result for regular languages: a reduction consisting of only program runs in FNF is ω-regular, optimal, and can be soundly proved terminating in place of the original program (Section 3).

To summarize, this paper proposes a way of improving termination checking for concurrent programs by exploiting commutativity to boost existing algorithmic verification techniques. We have implemented our proposed solution in a prototype termination checker for concurrent programs called TerMute, and present experimental results supporting the efficacy of the method in Section 6

2 Preliminaries

2.1 Concurrent Programs

In this paper, programs are *languages* over an alphabet of program statements Σ. The control flow graph for a *sequential* program with a set of locations Loc, and distinct entry and exit locations, naturally defines a finite automaton $(\mathsf{Loc}, \Sigma, \delta, \mathsf{entry}, \{\mathsf{exit}\})$. Without loss of generality, we assume that this automaton is deterministic and has a single exit location. This automaton recognizes *a language* of finite-length words. This is the set of all syntactic program runs that may or may not correspond to an actual program execution.

For the purpose of termination analysis, we are also interested in infinite-length program runs. Given a deterministic finite automaton $\mathcal{A}_L = (Q, \Sigma, \delta, q_0, F)$ with no dead states, where $\mathcal{L}(A_L) = L \subseteq \Sigma^*$ is a regular language of finite-length syntactic program runs, we define $\texttt{Büchi}(\mathcal{A}_L) = (Q, \Sigma, \delta, q_0, Q)$, a Büchi automaton recognizing the language $L^\omega = \{u \in \Sigma^\omega : \forall v \in pref(u). v \in pref(L)\}$, where $pref(u)$ denotes $\{w \in \Sigma^* : \exists w' \in \Sigma^* \cup \Sigma^\omega . w \cdot w' = u\}$ and $pref(L) = \bigcup_{v \in L} pref(v)$. These are all syntactic infinite program runs that may or may not correspond to an actual program execution.

We represent concurrency via interleaving semantics. A concurrent program is a parallel composition of a fixed number of threads, where each thread is a sequential program. Each thread P_i is recognized by an automaton $\mathcal{A}^i_{\mathsf{P}} = (\mathsf{Loc}_i, \Sigma_i, \delta_i, \mathsf{entry}_i, \{\mathsf{exit}_i\})$. We assume the Σ_i's are disjoint. The DFA recognizing $\mathsf{P} = \mathsf{P}_1 || \dots || \mathsf{P}_n$ is constructed using the standard product construction for a DFA $\mathcal{A}_\mathsf{P}$ recognizing the *shuffle* of the languages of the individual thread DFA's.

The language of infinite runs of this concurrent program, denoted P^ω, is the language recognized by $\texttt{Büchi}(\mathcal{A}_\mathsf{P})$. Note that a word in the language P^ω may not necessarily be the shuffle of infinite runs of its individual threads.

$$\mathsf{P}^\omega = \{u \in \Sigma^\omega |\ \exists i :\ u|_{\Sigma_i} \in \mathsf{P}^\omega_i \wedge \forall j :\ u|_{\Sigma_j} \in pref(\mathsf{P}_j) \cup \mathsf{P}^\omega_j\}$$

In the rest of the paper, we will simply write P when we mean P^ω for brevity. Note that P^ω includes *unfair* program runs, for example those in which individual threads can be indefinitely starved. As argued in [15], this can be easily fixed by intersecting P^ω with the set of all fair runs.

2.2 Termination

Let $\mathbb{X}$ the domain of the program state, Σ a set of statements, and denote $[\![.]\!] : \Sigma^* \to \mathcal{P}(\mathbb{X} \times \mathbb{X})$ a function which maps a sequence of statements to a relation over the program state, satisfying $[\![s_1]\!][\![s_2]\!] = [\![s_1 \cdot s_2]\!]$ for all $s_1, s_2 \in \Sigma^*$. Define sequential composition of relations in the usual fashion: $r_1 r_2 = \{(x, y) : \exists z.(x, z) \in r_1 \wedge (z, y) \in r_2\}$. We write $s(x)$ to denote $\{y : (x, y) \in [\![s]\!]\} \subseteq \mathbb{X}$.

We say that an infinite sequence of statements $\tau \in \Sigma^\omega$ is infeasible if and only if $\forall x \in \mathbb{X}\ \exists k \in \mathbb{N}\ s_1 \dots s_k(x) = \emptyset$, where s_i is the ith statement in the run τ. A program — an ω-regular language $P \subseteq \Sigma^\omega$ — is terminating if all of its *infinite* runs are infeasible.

$$\frac{\forall \tau \in P,\ \tau \text{ is infeasible}}{P \text{ is terminating}} \qquad (\text{TERM})$$

Lassos. It is not possible to effectively represent all infinite program runs, but we can opt for a slightly more strict rule by restricting our attention to ultimately periodic runs $UP \subseteq \Sigma^\omega$. That is, runs that are of the form uv^ω for some finite words $u, v \in \Sigma^*$. These are also typically called *lassos*.

It is *unsound* to replace *all runs* with *all ultimately periodic runs* in rule TERM. P may be non-terminating while all its ultimately periodic runs are

terminating. Assume that our program P is an ω-regular language and there is a universe $\mathcal{T}$ of known *terminating* programs that are all omega-regular languages. Then, we get the following *sound* rule instead:

$$\frac{\exists \Pi \in \mathcal{T}. P \subseteq \Pi}{P \text{ is terminating}} \qquad \text{(TermUP)}$$

If the inclusion $P \subseteq \Pi$ does not hold, then it is witnessed by an ultimately periodic run [4]. In a refinement loop in the style of Figure 2, one can iteratively expand Π based on this ultimately periodic witness (a.k.a. a lasso), and hence have a termination proof construction scheme in which ultimately periodic runs (lassos) are the only objects of interest. Note that if P includes unfair runs of a concurrent program, rather than fixing it, one can instead initialize Π with all the unfair runs of the concurrent program, which is an ω-regular language. This way, the rule becomes a fair termination rule.

2.3 Commutativity and Traces

An *independence* (or commutativity) relation $I \subseteq \Sigma \times \Sigma$ is a symmetric, anti-reflexive relation that captures the commutativity of a program's statements: $(s_1, s_2) \in I \implies [\![s_1 s_2]\!] = [\![s_2 s_1]\!]$. In what follows, assume such an I is fixed.

Finite Traces. Two finite words w_1 and w_2 are *equivalent* whenever we can apply a finite sequences of swaps of adjacent independent program statements to transform w_1 into w_2. Formally, an independence relation I on statements gives rise to an equivalence relation $\equiv_I$ on words by defining $\equiv_I$ to be the reflexive and transitive closure of the the relation $\sim_I$, defined as $us_1s_2v \sim_I us_2s_1v \iff (s_1, s_2) \in I$. A Mazurkiewicz trace $[u]_I = \{v \in \Sigma^* : v \equiv_I u\}$ is the corresponding equivalence class; we use "trace" exclusively to denote Mazurkiewicz traces.

Infinite Traces. Traces may also be defined in terms of dependence graphs (or partial orders). Given a word $\tau = s_1 s_2 ...$, the dependence graph corresponding to τ is a labelled, directed, acyclic graph $G = (V, E)$ with labelling function $L : V \to \Sigma$ and vertices $V = \{1, 2, \dots\}$, where $L(i) = s_i$, and $(i, i') \in E$ whenever $i < i'$ and $(L(i), L(i')) \notin I$. Then, $[\tau]_I^\infty$, the equivalence class of the infinite word τ, is precisely the set of *linear extensions* of G. Therefore, $\tau' \equiv_I \tau$ iff τ' is a linear extension of G.

For example, Figure 3(i) illustrates the Hasse diagram of the finite trace $[abcba]_I$, and Figure 3(ii), the Hasse diagram of the infinite trace $[abc(ab)^\omega]_I^\infty$, where $I = \{(a, b), (b, a)\}$.

For an infinite word τ, the *infinite trace* $[\tau]_I^\infty$ may contain linear extensions that do not

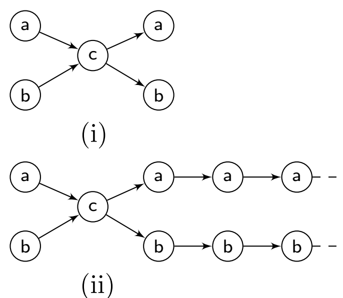


Fig. 3: Hasse diagrams.

correspond to any word in Σ^ω. For example, if $I = \{(a,b),(b,a)\}$, then the trace $[(ab)^\omega]_I^\infty$ includes a member (infinite word) in which all as appear before all bs. We use $a^\omega b^\omega$ to denote this word and call such words **transfinite**. This means that $[\tau]_I^\infty \not\subseteq \Sigma^\omega$, even for an ultimately periodic τ.

Normal Forms. A trace, as an equivalence class, may be represented unambiguously by one of its member words. *Lexicographical* normal forms [13] are the most commonly used normal forms, and the basis for the commonly known *sleep set* algorithm in partial order reduction [22]. *Foata Normal Forms* (FNF) are less well-known and are used in the technical development of this paper:

Definition 1 (Foata Normal Form of a finite trace [13]). *For a finite trace t, define FNF(t) as a sequence of sets $S_1 S_2 ... S_k$ (for some $k \in \mathbb{N}$) where $t = \Pi_i^k S_i$ and for all i:*

$$\forall a, b \in S_i \; a \neq b \implies (a,b) \in I \qquad \textit{(no dependencies in } S_i\textit{)}$$
$$\forall b \in S_{i+1} \; \exists a \in S_i \; (a,b) \notin I \qquad \textit{(}S_i \textit{ dependent on } S_{i+1}\textit{)}$$

Given a trace's dependence graphs, the FNF can be constructed by repeatedly removing sets of minimal elements, that is, sets with no incoming edges. Although we have defined a trace's FNF as a sequence of sets, we will generally refer to a trace's FNF as a word in which the elements in each set are assumed to be ordered lexicographically. For example, $\text{FNF}([abcba]_I) = ab \cdot c \cdot ab$, where $I = \{(a,b),(b,a)\}$. We overload this notation by writing $\text{FNF}([u]_I)$ as $\text{FNF}(u)$, and, for a language L, $\text{FNF}(L) = \{\text{FNF}(u) : u \in L\}$.

Theorem 1 ([13]). *L is a regular language iff the set of its Foata (respectively Lexicographical) normal forms is a regular language.*

3 Closures and Reductions

Commutativity defines an equivalence relation $\equiv_I$ which preserves the termination of a program run.

Proposition 1. *For $\tau, \tau' \in \Sigma^\omega$ and $\tau' \equiv_I \tau$, τ is terminating iff τ' is terminating.*

In the context of a refinement loop in the style of Figure 2, Proposition 1 suggests one can take advantage of commutativity by including all runs that are equivalent to the ones in Π (which are already proved terminating) in module (c). We formally discuss this strategy next.

Given a language L and an independence relation I, define $[L]_I^\infty = \cup_{\tau \in L} [\tau]_I^\infty$. Recall from Section 2 that, in general, $[\tau]_I^\infty \not\subseteq \Sigma^\omega$. Since programs are represented by ω-regular languages in our formalism, it is safe for us to exclude transfinite words from $[\tau]_I^\infty$ from commutativity closures computation. Define:

$$[L]_I^\omega = \cup_{\tau \in L} [\tau]_I^\infty \cap \Sigma^\omega \qquad (\omega\text{-closure})$$

The following sound proof rule is a straightforward adaptation of Rule TermUP that takes advantage of commutativity-based proof generalization:

$$\frac{\exists \Pi \subseteq \mathcal{T}.P \subseteq [\Pi]^{\omega}_{I}}{P \text{ is terminating}} \qquad \text{(TermClosure)}$$

Recall the example from Section 1 with two producers. The *transfinite* program run $\lambda_1^{\omega}\lambda_2^{\omega}$ that is the sequential compositions of the two producers looping forever back to back does not belong to the ω-closure of any ω-regular language. We generalize the notion of ω-closure to incorporate the idea of such runs in a new proof rule.

Let τ a transfinite word (like $a^{\omega}b^{\omega}$). Let τ' a prefix of τ. If $|\tau'| = \omega$, we say that τ' is an ω-prefix of τ, or $\tau' \in \mathit{pref}_{\omega}(\tau)$. A direct definition for when a transfinite word τ is terminating would be rather contrived, since a word such as $a^{\omega}b^{\omega}$ does not correspond to a program execution in the usual sense. However, a very useful property arises when considering the ω-words of $\mathit{pref}_{\omega}(\tau)$: If an ω-prefix τ' of a transfinite word τ is terminating, then all words in $[\tau]^{\omega}_{I}$ are terminating.

Theorem 2 (Omega Prefix Proof Rule). *Let $\tau'', \tau' \in \Sigma^{\omega}, \tau$ a transfinite word, if $\tau \equiv_I \tau''$ and $\tau' \in \mathit{pref}_{\omega}(\tau)$, τ' terminates $\Rightarrow \tau''$ terminates.*

Remark that $[\tau]^{\omega}_{I} \subseteq \Sigma^{\omega}$, so the former theorem uses the usual definition of termination, i.e. termination of ω-words; however; this theorem implicitly defines a notion of termination for some transfinite words.

Define $[\tau]^{p\omega}_{I}$, the *omega-prefix closure* of τ as

$$[\tau]^{p\omega}_{I} = [\tau]^{\omega}_{I} \cup \bigcup_{\tau'.\tau \in \mathit{pref}_{\omega}(\tau')} [\tau']^{\omega}_{I}.$$

Theorem 2 guarantees that, if τ terminates, then all of $[\tau]^{p\omega}_{I}$ terminates. The converse, however, does not necessarily hold: $[\tau]^{p\omega}_{I}$ is not an equivalence class.

Example 1. Continuing the example in Figure 1, recall that λ_1 and λ_2 are independent. Let us assume we have a proof that λ_1^{ω} is terminating. The class $[\lambda_1^{\omega}]^{\omega}_{I} = \{\lambda_1^{\omega}\}$ does not include any other members and therefore we cannot conclude the termination status of any other program runs based on it. On the other hand, since $\lambda_1^{\omega} \in \mathit{pref}_{\omega}(\lambda_1^{\omega}\lambda_2^{\omega})$ and $[(\lambda_1\lambda_2)^{\omega}]^{\omega}_{I} = [\lambda_1^{\omega}\lambda_2^{\omega}]^{\omega}_{I}$, $(\lambda_1\lambda_2)^{\omega} \in [\lambda_1^{\omega}]^{p\omega}_{I}$. Therefore, we can conclude that $(\lambda_1\lambda_2)^{\omega}$ is also terminating. Note that λ_2 can be non-terminating and the argument still stands.

One can replace the closure in Rule TermClosure with omega-prefix closure and produce a new, more powerful, sound proof rule. There is, however, a major obstacle in the way of an algorithmic implementation of Rule TermClosure with either closure scheme: the inclusion check in the premise is not decidable.

Proposition 2. *$[L]^{\omega}_{I}$ and $[L]^{p\omega}_{I}$ for an ω-regular language L may not be ω-regular. Moreover, it is undecidable to check the inclusions $L_1 \subset [L_2]^{\omega}_{I}$ and $L_1 \subset [L_2]^{p\omega}_{I}$ for ω-regular languages L_1 and L_2.*

3.1 The Compromise: A New Proof Rule

In the context of safety verification, with an analogous problem, a dual approach was proposed as a way forward [18] based on *program reductions.*

Definition 2 (ω-Reduction and ωp-Reduction). *A language $R \subseteq P$ is an ω-reduction (resp. ωp-reduction of P) of program P under independence relation I iff for all $\tau \in P$ there is some $\tau' \in R$ such that $\tau \in [\tau']_I^\omega$ (resp. $\tau \in [\tau']_I^{p\omega}$).*

The idea is that a program reduction can be soundly proven in place of the original program but, with strictly fewer behaviours to prove correct, less work has to be done by the prover.

Proposition 3. *Let P be a concurrent program and Π be ω-regular. We have:*

- *$P \subseteq [\Pi]_I^\omega$ iff there exists an ω-reduction R of P under I such that $R \subseteq \Pi$.*
- *$P \subseteq [\Pi]_I^{p\omega}$ iff there exists an ωp-reduction R of P under I such that $R \subseteq \Pi$.*

An $\omega/\omega p$-reduction R may not always be ω-regular. However, Proposition 3 puts forward a way for us to make a compromise to rule TermClosure for the sake of algorithmic implementability. Consider a *universe* of program reductions $\mathsf{Red}(P)$, which does not include *all* reductions. This gives us a new proof rule:

$$\frac{\exists \Pi \in \mathcal{T}. \exists R \in \mathsf{Red}(P). R \subseteq \Pi}{P \text{ is terminating}} \qquad (\textsc{TermReduc})$$

If $\mathsf{Red}(P)$ is the set of *all* ω-reductions (resp. ωp-reductions), then Rule TermReduc becomes logically equivalent to Rule TermClosure (resp. with $[\Pi]_I^{p\omega}$). By choosing a strict subset of all reductions for $\mathsf{Red}(P)$, we trade the undecidable premise check of the proof rule TermClosure with a new decidable premise check for Rule TermReduc. The specific algorithmic problem that this paper solves is then the following: What are good candidates for $\mathsf{Red}(P)$ such that an effective and efficient algorithmic implementation of Rule TermReduc exists? Moreover, we want this implementation to show significant advantages over the existing algorithms that implement the Rule TermUP.

In Section 5, we propose *Foata Reduction* as a theoretically clean option for $\mathsf{Red}(P)$ in the universe of all ω-reductions. In particular, they have the algorithmically essential property that the reductions do not include any transfinite words. In the universe of ωp-reductions, which does account for transfinite words, such a theoretically clean notion does not exist. This paper instead proposes the idea of mixing both closures and reductions as a best algorithmic solution for the undecidable Rule TermClosure in the form of the following new proof rule:

$$\frac{\exists \Pi \subseteq \mathcal{T}. \exists R \in \mathsf{Red}(P). R \subseteq [\Pi]_I^{opg}}{P \text{ is terminating}} \qquad (\textsc{TermOP})$$

In Section 3.2, we introduce $[\Pi]_I^{opg}$ as an underapproximation of $[\Pi]_I^{p\omega}$ that is guaranteed to be ω-regular and computable. Then, in Section 4, we discuss how, through a representation shift from infinite words to finite words, an appropriate class of reductions for $\mathsf{Red}(P)$ can be defined and computed.

3.2 Omega Prefix Generalization

We can implement the underapproximation of $[\Pi]_I^{p\omega}$ by generalizing the proof of termination of each individual lasso in the refinement loop of Figure 2. Let $u_1, ...u_m, v_1, ...v_{m'} \in \Sigma$ and consider the lasso uv^ω, where $u = u_1...u_m, v = v_1...v_{m'}$, and $m' > 0$. Let $\mathcal{A}_{uv^\omega} = (Q, \Sigma, \delta, q_0, \{q_m\})$ a Büchi automaton consisting of a stem and a loop, with a single accepting state q_m at the head of the loop, recognizing the ultimately periodic word uv^ω — in [25], this automaton is called a *lasso module* of uv^ω. Let $\Sigma_{I_{loop}} \subseteq \Sigma = \{a : \{v_1, ..., v_{m'}\} \times \{a\} \subseteq I\}$ the statements that are independent with the statements $v_1, \ldots, v_{m'}$ of the loop, and $\Sigma_{I_{stem}} \subseteq \Sigma_{I_{loop}} = \{a : \{u_1, \ldots, u_m, v_1, \ldots, v_{m'}\} \times \{a\} \subseteq I\}$ the statements that are independent of all statements appearing in uv^ω.

Define $\mathcal{OPG}(\mathcal{A}_\tau) = (Q \cup \{q'\}, \Sigma, \delta_{\mathcal{OPG}}, q_0, \{q_m\})$ for a lasso $\tau = uv^\omega$ where

$$\delta_{\mathcal{OPG}}(q, a) = \begin{cases} q & \text{if } q \in \{q_0, ..., q_{m-1}\} \wedge a \in \Sigma_{I_{stem}} \\ & \text{ or if } q \in \{q_{m+1}, ..., q_{m+m'}\} \cup \{q'\} \wedge a \in \Sigma_{I_{loop}} \\ q' & \text{if } q = q_m \wedge a \in \Sigma_{I_{loop}} \text{ or } m' = 1 \text{ and } a = v_1 \\ \delta(q_m, v_1) & \text{if } q = q' \wedge a = v_1 \\ \delta(q, a) & \text{o.w.} \end{cases}$$

We refer to the language $\mathcal{L}(\mathcal{OPG}(\mathcal{A}_\tau))$ recognized by this automaton as $[\tau]_I^{opg}$ for short. Note that this construction is given for individual lassos; we may generalize this to a (finite) set of lassos by simply taking their union. For a lasso $\tau = uv^\omega$, $\mathcal{OPG}(\mathcal{A}_\tau)$ is a linearly-sized Büchi automaton whose language satisfies the following:

Proposition 4. $[\tau]_I^{opg} \subseteq [\tau]_I^{p\omega}$.

Intuitively, this holds because this automaton simply allows us to intersperse the statements of uv^ω with independent statements; when considering the Mazurkiewicz trace arising from a word interspersed as described, these added independent statements may all be ordered after uv^ω, resulting in a transfinite word with ω-prefix uv^ω.

Theorem 3. *If τ is terminating, then every run in $[\tau]_I^{opg}$ is terminating.*

This follows directly from Theorem 2 and Proposition4, and concludes the soundness and algorithmic implementability of Rule TermOP if $\mathsf{Red}(P) = \{P\}$.

4 Finite-word Reductions

In this section, inspired by the program reductions used in safety verification, we propose a way of using those families of reductions to implement $\mathsf{Red}(P)$ in Rule TermReduc. This method can be viewed as a way of translating the liveness problem into an equivalent safety problem.

In [4], a finite-word encoding of ω-regular languages was proposed that can be soundly used for checking inclusion in the premise of rules such as Rule TermReduc:

Definition 3 (\$-language [4]). *Let $L \in \Sigma^\omega$. Define the \$-language of L as*

$$\$(L) = \{u\$v \mid u, v \in \Sigma^* \wedge uv^\omega \in L\}.$$

If L is ω-regular, then $\$(L)$ is regular [4]. This is proved by construction, but the one given in [4] is exponential. Since the Büchi automaton for a concurrent program P is already large, an exponential blowup to construct $\$(P)$ can hardly be tolerated. We propose an alternative polynomial construction.

4.1 Efficient Reduction to Safety

Our polynomial construction, denoted by fast\$, consists of linearly many copies of the Büchi automaton recognizing the program language.

Definition 4 (fast\$). *Given a Büchi automaton $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$, define fast\$($\mathcal{A}$) $= (Q_\$, \Sigma \cup \{\$\}, \delta_\$, q_0, F_\$)$ with $Q_\$ = Q \cup (Q \times Q \times \{0, 1\})$, $F_\$ = \{(q, q, 1) : q \in Q\}$, and for $q, r \in Q$, $i \in \{0, 1\}$:*

$$\delta_\$(q, a) = \begin{cases} \{(q, q, 0)\} & \text{if } a = \$ \\ \delta(q, a) & \text{o.w.} \end{cases}$$
$$\delta_\$((q, r, i), a) = \begin{cases} \{(q, r', 1) : r' \in \delta(r, a)\} & \text{if } i = 0 \text{ and } r \in F \\ \{(q, r', i) : r' \in \delta(r, a)\} & \text{o.w.} \end{cases}$$

Let L be an ω-regular language and $\mathcal{A}$ be a Büchi automaton recognizing L. We overload the notation and use fast\$(L) to denote the language recognized by fast\$($\mathcal{A}$). Note that fast\$(L), unlike $\$(L)$, is a construction parametric on the Büchi automaton recognizing the language, rather than the language itself. In general, fast\$(L) under-approximates $\$(L)$. But, under the assumption that all alphabet symbols of Σ label at most one transition in the Büchi automaton $\mathcal{A}$ (recognizing L), then fast\$(L) $= \$(L)$. This condition is satisfied for any Büchi automaton that is constructed from the control flow graph of a (concurrent) program since we may treat each statement appearing on the graph as unique, and these graph edges correspond to the transitions of the automaton.

Theorem 4. *For any ω-regular language L, we have fast\$(L) $\subseteq \$(L)$. If P is a concurrent program then fast\$(P) $= \$(P)$.*

First, let us observe that in Rule TermUP, we can replace P with fast\$(P) and Π with fast\$(Π) (and hence the universe $\mathcal{T}$ with a correspondingly appropriate universe) and derive a new *sound* rule.

Theorem 5. *The finite word version of Rule* TermUP *using fast\$ is sound.*

The proof of Theorem 5 follows from Theorem 4. Using fast\$, the program is precisely represented and the proof is under-approxiamted, therefore the inclusion check implies the termination of the program.

4.2 Sound Finite Word Reductions

With a finite word version of the Rule TermUP, the natural question arises if one can adopt the analogue of the sound proof rule used for safety [18] by introducing an appropriate class of reductions for program termination in the following proof rule:

$$\frac{\exists \Pi \in \mathcal{T}. \exists R \in \mathsf{Red}(\$(P)). R \subseteq \mathsf{fast\$}(\Pi)}{P \text{ is terminating}} \qquad (\textsc{FiniteTermReduc})$$

A language R is a *sound reduction* of $\$(P)$ if the termination of all ultimately periodic words uv^ω, where $u\$v \in R$, implies the termination of all ultimately periodic words of P. Since, in $u\$v$, the word u represents the stem of a lasso and the word v represents its loop, it is natural to define equivalence, considering the two parts separately, that is: $u\$v \equiv_I u'\$v'$ iff $u' \equiv_I u \wedge v' \equiv_I v$. One can use any technique for producing reductions for safety, for example *sleep sets* for lexicographical reductions [18], in order to produce a *sound* reduction that includes representatives from this equivalence relation. Assume that \$ does not commute with any other letter in an extension $I_\$$ of I over $\Sigma \cup \{\$\}$ and observe that the standard finite-length word Mazurkiewicz equivalence relation of $u\$v \equiv_{I_\$} u'\$v'$ coincides with $u\$v \equiv_I u'\$v'$ as defined above. Let $\mathit{FRed}(\$(P))$ be the set of all such reductions. An algorithmic implementation of Rule FiniteTermReduc with $\mathsf{Red}(\$(P)) = \mathit{FRed}(\$(P))$ may then be taken straightforwardly from the safety literature.

Note, however, that reductions in $\mathit{FRed}(\$(P))$ are more restrictive than their infinite analogues; for example, $uv\$v \notin [u\$v]_I$, whereas $uvv^\omega = uv^\omega$ and therefore $uvv^\omega \equiv_I uv^\omega$ for any I. By treating $\$(P)$'s \$-word as a a finite word without recognizing its underlying lasso structure, every word uv^ω in the program necessarily engenders an infinite family of representatives in R — one for each \$-word $\{u\$v, uv\$v, u\$vv, ...\} \subseteq \$(P)$ corresponding to $uv^\omega \in P$.

We define *dollar closure* as variant of classic closure that is sensitive to the termination equivalence of the corresponding infinite words:

$$[u\$v]_I^\$ = \{x\$y : uv^\omega \in [xy^\omega]_I^{p\omega}\}$$

The termination of uv^ω is implied by the termination of any xy^ω such that $x\$y$ is a member of $[u\$v]_I^\$$ (see Theorem 2). However, the converse does not necessarily hold. Therefore, like omega-prefix closure, $[u\$v]_I^\$$ is not an equivalence class. It suggests a more relaxed condition (than the one used for $\mathit{FRed}(\$(P))$) for the soundness of a reduction:

Definition 5 (Sound \$-Program Reduction). *A language $R \subseteq P$ is called a* sound *\$-program reduction of $\$(P)$ under independence relation I iff for all $uv^\omega \in P$ we have $[u\$v]_I^\$ \cap R \neq \emptyset$.*

A \$-reduction R satisfying the above condition is obviously sound: It must contain a \$-representative $x\$y \in [u\$v]_I^\$$ for each word uv^ω in the program. If R is terminating, then xy^ω is terminating, and therefore so is uv^ω. Moreover, these

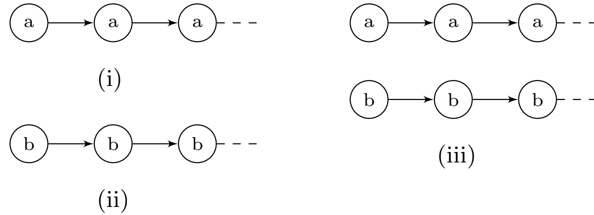


Fig. 4: The only three traces in $P = a^\omega || b^\omega$ when $(a, b) \in I$.

sound \$-program reductions can be quite parsimonious, since one word can be an omega-prefix corresponding to many classes of program behaviours.

Under this soundness condition, we may now include one representative of $[u\$v]_I^\$$ for each $uv^\omega \in P$ in a sound reduction of P. For example, $R = \{\$a, \$b\}$ is a sound \$-program reduction of $P = a^\omega || b^\omega$ when $(a, b) \in I$. To illustrate, note that the only traces of P are the three depicted as Hasse diagrams in Figure 4; the distinct program words $(ab)^\omega, (aba)^\omega, (abaa)^\omega, \ldots$ all correspond to the same infinite trace shown in Figure 4(iii). A salient feature of Figure 4(iii) is that a^ω and b^ω correspond to *disconnected* components of this dependence graph. The omega-prefix rule of Theorem 2 can be interpreted in this graphical context as follows: if any *connected component* of the trace is terminating, then the entire class is terminating.

Recall that module (d) of the refinement loop of Figure 2 may naturally be implemented as the inclusion check $P \subseteq \Pi$, or one of its variations that appear in the proof rules proposed throughout this paper. In a typical inclusion check, a product of the program and the complement of the proof automata are explored for the reachability of an accept state. Therefore, classic reduction techniques that operate on the program by pruning transitions/states during this exploration are highly desirable in this context. We propose a repurposing of such techniques that shares the simplicity and efficiency of constructing reductions from $FRed(\$(P)))$ (in the style of safety) and yet takes advantage of the weaker soundness condition in Definition 5 and performs a more aggressive reduction. In short, a reduced program may be produced by pruning transitions while performing an on-the-fly exploration of the program automaton. In pruning, our goal is to discard transitions that would necessarily form words whose suffixes lead us into the disconnected components of the program traces underlying the program words that have been explored so far. This selective pruning technique is provided by a straightforward adaptation of the well-known safety reduction technique of persistent sets [22]. Consider the program illustrated in Figure 5(a). In the graph in Figure 5(b), the green states are explored and the dashed transitions are pruned. This amounts to proving two lassos terminating in the refinement loop of Figure 2, where each lasso corresponds to one connected component of a program trace.

We compute persistent sets using a variation of Algorithm 1 in Chapter 4 of [22]. In brief, $a \in Persistent_\prec(q)$ if a is the lexicographically least enabled state at q according to thread order $\prec$, if a is an enabled statement from

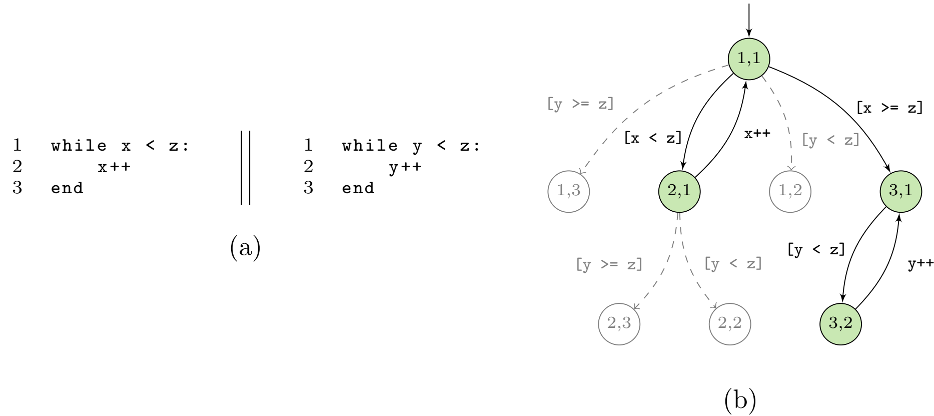


Fig. 5: Example of persistent set selective search.

the same thread as another statement $a' \in Persistent_{\prec}(q)$, or if a is dependent on some statement $a' \in Persistent_{\prec}(q)$ from a different thread than a. In addition, \$ is also persistent whenever it is enabled. This set may be computed via a fixed-point algorithm; whenever a statement that is not enabled is added to $Persistent_{\prec}(q)$, then $Persistent_{\prec}(q)$ is simply the set of all enabled states. Intuitively, this procedure works because transitions are ignored only when they are necessarily independent from all the statements that will be explored imminently; these may soundly be ignored indefinitely or deferred. Transitions that are deferred indefinitely are precisely those that would lead into a disconnected component of a program traces.

Algorithm 1: PersistentSS

Input: $\mathsf{fast\$}(\mathcal{A}_P) = (Q, \Sigma, \delta, q_0, F)$
Output: $x\$y$
1 $H \leftarrow \emptyset, S \leftarrow \{(q_0, \text{“”})\}$
2 **while** $(q, w) = S.pop()$ **do**
3   **if** $q \notin H$ **then**
4   **if** $q \in F$ **then**
5   **return** w
6   **for** $a \in \Sigma \cap Persistent(q)$ **do**
7             $S.push(\delta(q, a), w \cdot a)$
8         $H \leftarrow H \cup \{q\}$
9 **return** *“EMPTY”*

The reduced program that arises from the persistent set selective search of $\mathsf{fast\$}(\mathcal{A}_P)$ based on thread order $\prec$ is denoted by $\mathtt{PersistentSS}_{\prec}(\$(P))$. Figure 5(b) illustrates a reduced program; note that \$-transitions are omitted for simplicity. The reduced program corresponds to the states shown in green. The other program states are unreachable because the only persistent transitions correspond to statements from the least enabled thread; the transitions shown with dashed lines are not persistent.

Theorem 6 (soundness of finite word reductions). *Rule* FiniteTermReduc *is a sound proof rule when* $\mathsf{Red}(\$(P)) = \{\forall \prec\colon \mathtt{PersistentSS}_{\prec}(\$(P))\}$.

The theorem holds under the condition that the set $\mathcal{T}$ from Rule FiniteTermReduc is the set of all terminating ω-regular languages, and the under the assumption that the program is fair (or, equivalently, that the proof includes the unfair

runs of P, as discussed in Section 2.2), where a fair run is one where no enabled thread action is indefinitely deferred. The proof of soundness appears in the extended version of this paper [31]. Intuitively, it relies on the fact that $\texttt{PersistentSS}_{\prec}(\$(P))$ is a $-*program reduction* for all the fair runs in P.

Example 2. Recall the producer-consumer in Figure 1, and consider the program with two producers P_1 and P_2 and one consumer C. Let λ_1 denote the loop body of P_1, and λ_2 that of P_2. Concretely, $\lambda_1 =$ `[i < producer_limit] ; C++ ; i++` where `[...]` is an *assume* statement, and similarly for λ_2. In addition, each loop has an exit statement, which we denote by ι_1 and ι_2. For instance, $\iota_1 =$ `[i >= producer_limit]`. Let $\prec$ such that $P_1 \prec P_2 \prec C$.

In $\mathcal{A} = \texttt{PersistentSS}_{\prec}(\$(P))$, P_1 is the *first* thread and therefore persistent; that is, the word $\$\lambda_1$ — the $-word corresponding to λ_1^ω – is in the reduction. Since λ_1 is independent of all statements in P_2 and C, any run in which P_1 enters the loop (and does not exit via ι_1) will not be included in the reduction. In effect, this means that λ_1^ω is the only representative of $[\lambda_1^\omega]_I^{p\omega} = [\lambda_1^\omega]_I^\omega \cup [\lambda_1^\omega \cdot (P_2 + C)^\omega]_I^\omega$ in the program reduction.

Even though P_2 seems identical to P_1, the same is not true for P_2 because it appears later in the thread order. In this case, $[\lambda_2]_I^{p\omega}$ is represented by the family of words $(\lambda_1)^*\iota_1\lambda_2^\omega$.

5 Omega Regular Reductions

In the classic implementation of Rule TermUP [25], ω-regular languages are used to represent the program P and the proof Π. It is therefore natural to ask if $\mathsf{Red}(P)$ in Rule TermReduc can be a family of ω-regular program reductions. For *finite* program reductions [18,19,16,17], and also for classic POR, lexicographical normal forms are almost always the choice. *Infinite traces* have lexicographic normal forms that are analogous to their finite counterparts [13]. However, these normal forms are not suitable for defining $\mathsf{Red}(P)$. For example, if $(a,b) \in I$, then the lexicographic normal form of the trace $[(ab)^\omega]_I^\infty$ is $a^\omega b^\omega$ if $a < b$ or $b^\omega a^\omega$ otherwise; both transfinite words. Fortunately, Foata normal forms do not share the same problem.

Definition 6 (Foata Normal Form of an infinite trace [13]). *Foata Normal Form FNF(t) of an infinite trace t is a sequence of non-empty sets $S_1 S_2 ...$ such that $t = \Pi_{i \le \omega} S_i$ and for all i:*

$$\forall a, b \in S_i \;\; a \neq b \implies (a,b) \in I \qquad \textit{(no dependencies in } S_i\textit{)}$$
$$\forall b \in S_{i+1} \; \exists a \in S_i \;\; (a,b) \notin I \qquad (S_i \textit{ dependent on } S_{i+1}\textit{)}$$

For example, $\mathrm{FNF}([(ab)^\omega]_I^\infty) = (ab)^\omega$ if $(a,b) \in I$. To define a reduction based on FNF, we need a mild assumption about the program language.

Definition 7 (Closedness). *A language $L \subseteq \Sigma^\infty$ is* closed *under the independence relation I iff $[L]_I^\infty \subseteq L$ and is ω*-closed *under I iff $[L]_I^\omega \subseteq L$.*

It is straightforward to see that any concurrent program P (as defined in Section 2.1), and any valid dependence relation I, we have that P is ω-closed. This means that for any (infinite) program run τ, any other ω-word τ that is equivalent to τ is also in the language of the program.

The key result that makes Foata normal forms amenable to automation in the automaton-theoretic framework is the following theorem.

Theorem 7. *If $L \subseteq \Sigma^\omega$ is ω-regular and closed, FNF(L) is ω-regular.*

The proof of this theorem provides a construction for the Büchi automaton that recognizes the language FNF(L); see [31] for more detail. However, this construction is not efficient since, for a program P, of size $\Theta(n)$, the Büchi automaton recognizing FNF(P) can be as large as $\mathcal{O}(n2^n)$. Finally, Foata reductions are *minimal* in the same exact sense that lexicographical reductions of finite-word languages are minimal:

Theorem 8. *[Theorem 11.2.15 [13]] If $L \subseteq \Sigma^\omega$ is ω-regular and closed, then for all $\tau \in L$, $\tau' \in FNF(L) \cap [\tau]_I^\omega \implies \tau' = \tau$.*

Our experimental results in Section 6 suggest that this complexity is a big bottleneck in practical benchmarks. Therefore, despite the fact that Foata normal forms put forward an algorithmic solution for the implementation of Rule TermReduc, the inefficiency of the solution makes it unsuitable for practical termination checkers.

6 Experimental Results

The techniques presented in this paper have been implemented in a prototype tool called TerMute written in Python and C++. The inputs are concurrent integer programs written in a C-like language. TerMute may output "Terminating", or "Unknown", in the latter case also returning a lasso whose termination could not be proved. Ranking functions and invariants are produced using the method described in [24], which is restricted to linear ranking functions of linear lassos. Interpolants are generated using SMTInterpol [6] and MathSAT [7]; the validity of Hoare triples are checked using CVC4 [2].

TerMute may be run in several different modes. FOATA is an implementation of the algorithm described in Section 5. The baseline is the core counterexample-guided refinement algorithm of [25], which has been adapted to the finite-word context in order to operate on the automata fast\$$(P)$ and fast\$$(\Pi)$ of Section 4.1. All other modes are modifications of this baseline, maintaining the same refinement scheme, so that we can isolate the impact of adding commutativity reasoning. Hoare triple generalization ("HGen") augments the baseline by making solver calls after each refinement round in order to determine if edges may soundly be added to the proof for any valid Hoare triples not produced as part of the original proof. "POR" implements the persistent set technique of Section 4.2 and "OPG" is the finite-word analogue of the ω-prefix

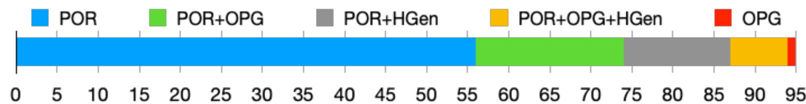
generalization in Section 3.2. TerMute can also be run on any combinations of these techniques. In what follows, we use TerMute to refer to the portfolio winner among all algorithms that employ *commutativity reasoning*, namely POR, OPG, POR + HGen, POR + OPG, and POR + OPG + HGen.

See [31] for more detail regarding our experimental setup and results.

Benchmarks. Our benchmarks include 114 terminating concurrent linear integer programs that range from 2 to 12 threads and cover a variety of patterns commonly used for synchronization, including the use of locks, barriers, and monitors. Some are drawn from the literature on termination verification of concurrent programs, specifically [29,34,37], and the rest were created by us, some of which are based on sequential benchmarks from The Termination Problem Database [38], modified to be multi-threaded. We include programs whose threads display a wide range of independence — from complete independence (e.g. the producer threads in Figure 1), all the way to complete dependence — and demonstrate a range of complexity with respect to control flow.

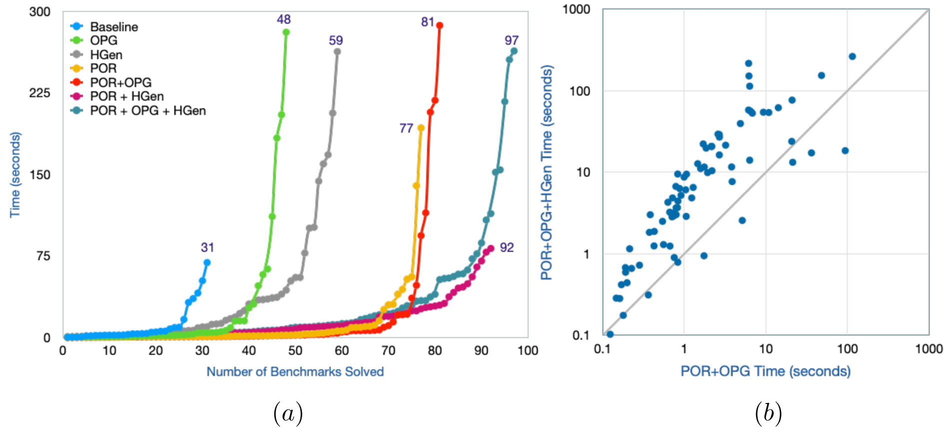
Results. Our experiments have a timeout of 300 seconds and a memory cap of 32 GB, and were run on a 12th Gen Intel Core i7-12700K with 64 GB of RAM running Ubuntu 22.04. We experimented with both interpolating solvers and the reported times correspond to the winner of the two. The results are depicted in Figure 6(a) as a *quantile* plot that compares the algorithms. The total number of benchmarks solved is noted on each curve. FOATA times out on all but the simplest benchmarks, and therefore is omitted from the plot.

The portfolio winner, TerMute, solves 101 benchmarks in total. It solves any benchmark otherwise solved by algorithms without commutativity reasoning (namely, the baseline or HGen). It is also faster on 95 out of 101 benchmarks it solves. The figure below illustrates how often each of the portfolio algorithms emerges as the fastest among these 95 benchmarks.



HGen aggressively generalizes the proof and consequently, it forces convergence in many fewer refinement rounds. This, however, comes at the cost of a time overhead per round. Therefore, HGen helps in solving more benchmarks, but whenever a benchmarks is solvable without it, it is solved much faster. The scatter plot in Figure 6(b) illustrates this phenomenon when HGen is added to POR+OPG. The plot compares the times of benchmarks solved by both algorithms on a logarithmic scale, and the overhead caused by HGen is significant in the majority of the cases.

Recall, from Section 4, that the persistent set algorithm is parametrized on an order over the participating threads. The choice of order centrally affects the way the persistent set algorithm works, by influencing which transitions may



(a) (b)

Fig. 6: Experimental results for TerMute: (a) quantile plot for the throughput of each algorithm, and (b) scatter plot for the impact of thread order on efficiency.

be explored and, by extension, which words appear in the reduced program. Experimentally, we have observed that the chosen order plays a significant role in how well the algorithms work, but to varying degrees. For instance, for POR, the worst thread order times out on 16% of the benchmarks that the best order solves. For POR+OPG+HGen, the difference is more modest at 7%. In practice, it is sensible then to instantiate a few instances of the TerMute with a few different random orders to increase the chances of getting better performance.

7 Related Work

The contribution of this paper builds upon sequential program termination provers to produce termination proofs for concurrent programs. As such, any progress in the state of the art in sequential program termination can be used to produce proofs for more lassos, and is, therefore, complementary to our approach. So, we only position this paper in the context of algorithmic concurrent program termination, and the use of commutativity for verification in general, and skip the rich literature on sequential program termination [36,11] or model checking liveness [8,26,9,33].

Concurrent Program Termination. The thread-modular approach to proving termination of concurrent programs [34,10,37,35] aims to prove a thread's termination without reasoning directly about its interactions with other threads, but rather by inferring facts about the thread's environment. In [37], this approach is combined with compositional reasoning about termination arguments. Our technique can also be viewed as modular in the sense that lassos – which, like isolated threads, are effectively sequential programs – are dealt with independently of the broader program in which they appear; however, this is distinct from thread-modularity insofar as we reason directly about behaviours arising

from the interaction of threads. Whenever a thread-modular termination proof can be automatically generated for the program, that proof is the most efficient in terms of scalability with the number of threads. However, for a thread-modular proof to always exist, local thread states have to be exposed as auxiliary information. The modularity in our technique does not rely on this information at all. Commutativity can be viewed as a way of observing and taking advantage of some degree of non-interference, different from that of thread modularity.

Causal dependence [29] presents an abstraction refinement scheme for proving concurrent programs terminating that takes advantage of the equivalence between certain classes of program runs. These classes of runs are determined by partial orders that capture the causal dependencies between transitions, in a manner reminiscent of the commutativity-based partial orders of Mazurkiewicz traces. The key to scalability of this method is that they forgo a containment check in the style of module (d) of Figure 2. Instead, they cover the space of program behaviour by splitting it into cases. Therefore, for the producer-only instance of the example in Section 1, this method can scale to many many thread easily, while our commutativity-based technique cannot. Similar to thread-modular approach, this technique cannot be beaten in scalability for the programs that can be split into linearly many cases. However, there is no guarantee (none given in [29]), that a bounded complete causal trace tableau for a terminating program must exist; for example, when there is a dependency between loops in different threads that would cause the program to produce unboundedly many (Mazurkiewicz) traces that have to be analyzed for termination. The advantage of our method is that, once consumers are added to the example in Section 1, we can still take advantage of all the existing commutativity to gain more efficiency.

Similar to safety verification, context bounding [3] has been used as a way of under-approximating concurrent programs for termination analysis as well.

Commutativity in Verification. Program reductions have been used as a means of simplifying proofs of concurrent and distributed programs before. Lipton's movers [32] have been used to simplify programs for verification. CIVL [28,27] uses a combination of abstraction and reduction to produce *layered programs*; in an interactive setup, the programmer can prove that an implementation satisfies a specification by moving through these layered programs to increasingly more abstract programs. In the context of message-passing distributed systems [12,21], commutativity is used to produce a synchronous (rather than sequential) program with a simpler proof of correctness.

In [18,19,16,17] program reductions are used in a refinement loop in the same style as this paper to prove safety properties of concurrent programs. In [18,19], an unbounded class of lexicographical reductions are enumerated with the purpose of finding a simple proof for at least one of the reductions; the thesis being that there can be a significant variation in the simplicity of the proof for two different reductions. In [19], the idea of contextual commutativity — i.e. considering two statements commutative in some context yet not all contexts — is introduced and algorithmically implemented. In [16,17], only one reduction

at a time is explored, in the same style as this paper. In [16], a persistent-set-based algorithm is used to produce space-efficient reductions. In [17] the idea of abstract commutativity is explored. It is shown that no *best* abstraction exists that provides a maximal amount of commutativity and, therefore, the paper proposes a way to combine the benefits of different commutativity relations in one verification algorithm. The algorithm in this paper can theoretically take advantage of all of these (orthogonal) findings to further increase the impact of commutativity in proving termination.

Non-termination. The problem of detecting non-termination has also been directly studied [5,23,20,1,30]. Presently, our technique does not accommodate proving the non-termination of a program. However, it is relatively straightforward to adapt any such technique (or directly use one of these tools) to accommodate this; in particular, when we fail to find a termination proof for a particular lasso, sequential methods for proving non-termination may be employed to determine if the lasso is truly a non-termination witness. However, it is important to note that a program may be non-terminating while all its lassos are terminating, and the refinement loop in Figure 2 may just diverge without producing a counterexample in this style; this is a fundamental weakness of using lassos as modules to prove termination of programs.

8 Conclusion

In the literature on the usage of commutativity in safety verification, sound program reductions are constructed by selecting lexicographical normal forms of equivalence classes of concurrent program runs. These are not directly applicable in the construction of sound program reductions for termination checking, since the lexicographical normal forms of infinite traces may not be ω-words. In this paper, we take this apparent shortcoming and turn it into an effective solution. First, these transfinite words are used in the design of the *omega prefix proof rule* (Theorem 2). They also inform the design of the *termination aware* persistent set algorithm described in Section 4.2. Overall, this paper contributes mechanisms for using commutativity-based reasoning in termination checking, and demonstrates that, using these mechanisms, one can efficiently check the termination of concurrent programs.

References

1. Atig, M.F., Bouajjani, A., Emmi, M., Lal, A.: Detecting fair non-termination in multithreaded programs. In: Computer Aided Verification: 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings 24. pp. 210–226. Springer (2012)
2. Barrett, C.W., Conway, C.L., Deters, M., Hadarean, L., Jovanovic, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird,

UT, USA, July 14-20, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6806, pp. 171–177. Springer (2011). https://doi.org/10.1007/978-3-642-22110-1_14, https://doi.org/10.1007/978-3-642-22110-1_14

3. Baumann, P., Majumdar, R., Thinniyam, R.S., Zetzsche, G.: Context-bounded verification of liveness properties for multithreaded shared-memory programs. Proceedings of the ACM on Programming Languages **5**(POPL), 1–31 (2021)
4. Calbrix, H., Nivat, M., Podelski, A.: Ultimately periodic words of rational ω-languages. In: International Conference on Mathematical Foundations of Programming Semantics. pp. 554–566. Springer (1993)
5. Chatterjee, K., Goharshady, E.K., Novotnỳ, P., Žikelić, Đ.: Proving non-termination by program reversal. In: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation. pp. 1033–1048 (2021)
6. Christ, J., Hoenicke, J., Nutz, A.: Smtinterpol: An interpolating smt solver. In: Model Checking Software: 19th International Workshop, SPIN 2012, Oxford, UK, July 23-24, 2012. Proceedings 19. pp. 248–254. Springer (2012)
7. Cimatti, A., Griggio, A., Schaafsma, B.J., Sebastiani, R.: The mathsat5 smt solver. In: Tools and Algorithms for the Construction and Analysis of Systems: 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings 19. pp. 93–107. Springer (2013)
8. Cohen, A., Namjoshi, K.S.: Local proofs for linear-time properties of concurrent programs. In: Computer Aided Verification: 20th International Conference, CAV 2008 Princeton, NJ, USA, July 7-14, 2008 Proceedings 20. pp. 149–161. Springer (2008)
9. Cook, B., Koskinen, E., Vardi, M.Y.: Temporal property verification as a program analysis task. In: Gopalakrishnan, G., Qadeer, S. (eds.) Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6806, pp. 333–348. Springer (2011)
10. Cook, B., Podelski, A., Rybalchenko, A.: Proving thread termination. In: Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 320–330 (2007)
11. Cook, B., Podelski, A., Rybalchenko, A.: Proving program termination. Commun. ACM **54**(5), 88–98 (2011)
12. Desai, A., Garg, P., Madhusudan, P.: Natural proofs for asynchronous programs using almost-synchronous reductions. SIGPLAN Not. **49**(10), 709–725 (oct 2014). https://doi.org/10.1145/2714064.2660211, https://doi.org/10.1145/2714064.2660211
13. Diekert, V., Rozenberg, G.: The book of traces. World scientific (1995)
14. Elmas, T., Qadeer, S., Tasiran, S.: A calculus of atomic actions. In: Shao, Z., Pierce, B.C. (eds.) Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009. pp. 2–15. ACM (2009)
15. Farzan, A., Kincaid, Z., Podelski, A.: Proving liveness of parameterized programs. In: Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science. p. 185–196. LICS '16, Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2933575.2935310, https://doi.org/10.1145/2933575.2935310

16. Farzan, A., Klumpp, D., Podelski, A.: Sound sequentialization for concurrent program verification. In: Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation. pp. 506–521 (2022)
17. Farzan, A., Klumpp, D., Podelski, A.: Stratified commutativity in verification algorithms for concurrent programs. Proceedings of the ACM on Programming Languages **7**(POPL), 1426–1453 (2023)
18. Farzan, A., Vandikas, A.: Automated hypersafety verification. In: Computer Aided Verification: 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I 31. pp. 200–218. Springer (2019)
19. Farzan, A., Vandikas, A.: Reductions for safety proofs. Proceedings of the ACM on Programming Languages **4**(POPL), 1–28 (2019)
20. Frohn, F., Giesl, J.: Proving non-termination via loop acceleration. arXiv preprint arXiv:1905.11187 (2019)
21. v. Gleissenthall, K., Kıcı, R.G., Bakst, A., Stefan, D., Jhala, R.: Pretend synchrony: Synchronous verification of asynchronous distributed programs. Proc. ACM Program. Lang. **3**(POPL) (jan 2019). https://doi.org/10.1145/3290372, https://doi.org/10.1145/3290372
22. Godefroid, P.: Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem. Springer (1996)
23. Gupta, A., Henzinger, T.A., Majumdar, R., Rybalchenko, A., Xu, R.G.: Proving non-termination. In: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 147–158 (2008)
24. Heizmann, M., Hoenicke, J., Leike, J., Podelski, A.: Linear ranking for linear lasso programs. In: Automated Technology for Verification and Analysis: 11th International Symposium, ATVA 2013, Hanoi, Vietnam, October 15-18, 2013. Proceedings. pp. 365–380. Springer (2013)
25. Heizmann, M., Hoenicke, J., Podelski, A.: Termination analysis by learning terminating programs. In: Computer Aided Verification: 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings 26. pp. 797–813. Springer (2014)
26. Koskinen, E., Terauchi, T.: Local temporal reasoning. In: Henzinger, T.A., Miller, D. (eds.) Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), CSL-LICS '14, Vienna, Austria, July 14 - 18, 2014. pp. 59:1–59:10. ACM (2014)
27. Kragl, B., Enea, C., Henzinger, T.A., Mutluergil, S.O., Qadeer, S.: Inductive sequentialization of asynchronous programs. In: Donaldson, A.F., Torlak, E. (eds.) Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020. pp. 227–242. ACM (2020)
28. Kragl, B., Qadeer, S.: Layered concurrent programs. In: Chockler, H., Weissenbacher, G. (eds.) Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I. Lecture Notes in Computer Science, vol. 10981, pp. 79–102. Springer (2018)
29. Kupriyanov, A., Finkbeiner, B.: Causal termination of multi-threaded programs. In: Computer Aided Verification: 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings 26. pp. 814–830. Springer (2014)

30. Le, T.C., Antonopoulos, T., Fathololumi, P., Koskinen, E., Nguyen, T.: Dynamite: dynamic termination and non-termination proofs. Proc. ACM Program. Lang. **4**(OOPSLA), 189:1–189:30 (2020)
31. Lette, D., Farzan, A.: Commutativity for concurrent program termination proofs (extended version) https://www.cs.toronto.edu/~azadeh/resources/papers/cav23-extended.pdf
32. Lipton, R.J.: Reduction: A method of proving properties of parallel programs. Commun. ACM **18**(12), 717–721 (1975)
33. Liu, Y.C., Pang, C., Dietsch, D., Koskinen, E., Le, T., Portokalidis, G., Xu, J.: Proving LTL properties of bitvector programs and decompiled binaries. In: Oh, H. (ed.) Programming Languages and Systems - 19th Asian Symposium, APLAS 2021, Chicago, IL, USA, October 17-18, 2021, Proceedings. Lecture Notes in Computer Science, vol. 13008, pp. 285–304. Springer (2021)
34. Malkis, A., Podelski, A., Rybalchenko, A.: Precise thread-modular verification. In: Nielson, H.R., Filé, G. (eds.) Static Analysis, 14th International Symposium, SAS 2007, Kongens Lyngby, Denmark, August 22-24, 2007, Proceedings. Lecture Notes in Computer Science, vol. 4634, pp. 218–232. Springer (2007)
35. Pani, T., Weissenbacher, G., Zuleger, F.: Rely-guarantee bound analysis of parameterized concurrent shared-memory programs: With an application to proving that non-blocking algorithms are bounded lock-free. Formal Methods in System Design **57**(2), 270–302 (2021)
36. Podelski, A., Rybalchenko, A.: Transition invariants. In: 19th IEEE Symposium on Logic in Computer Science (LICS 2004), 14-17 July 2004, Turku, Finland, Proceedings. pp. 32–41. IEEE Computer Society (2004)
37. Popeea, C., Rybalchenko, A.: Compositional termination proofs for multi-threaded programs. In: Tools and Algorithms for the Construction and Analysis of Systems: 18th International Conference, TACAS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24–April 1, 2012. Proceedings 18. pp. 237–251. Springer (2012)
38. The termination problem database (2023), https://github.com/TermCOMP/TPDB