A Framework to Synergize Partial Order Reduction with State Interpolation

Duc-Hiep Chu and Joxan Jaffar

National University of Singapore
hiepcd,joxan@comp.nus.edu.sg

Abstract. We address the problem of reasoning about interleavings in safety verification of concurrent programs. In the literature, there are two prominent techniques for pruning the search space. First, there are well-investigated *trace-based* methods, collectively known as "Partial Order Reduction (POR)", which operate by weakening the concept of a trace by abstracting the total order of its transitions into a partial order. Second, there is *state-based* interpolation where a collection of formulas can be generalized by taking into account the property to be verified. Our main contribution is a framework that *synergistically* combines POR with state interpolation so that the sum is more than its parts.

1 Introduction

We consider the *state explosion problem* in safety verification of concurrent programs. This is caused by the interleavings of transitions from different processes. In explicit-state model checking, a general approach to counter this explosion is Partial Order Reduction (POR) (e.g., [23, 10]). This exploits the equivalence of interleavings of "independent" transitions: two transitions are independent if their consecutive occurrences in a trace can be swapped without changing the final state. In other words, POR-related methods prune away *redundant* process interleavings in a sense that, for each Mazurkiewicz [18] trace equivalence class of interleavings, if a representative has been checked, the remaining ones are regarded as redundant.

On the other hand, *symbolic execution* [17] is another method for program reasoning which recently has made increasing impact on software engineering research [3]. The main challenge for symbolic execution is the exponential number of symbolic paths. The works [15, 19, 14] tackle successfully this fundamental problem by eliminating from the concrete model, on-the-fly, those facts which are *irrelevant* or *too-specific* for proving the unreachability of the error nodes. This learning phase consists of computing *state-based interpolants* in a similar spirit to that of conflict clause learning in SAT solvers.

Now symbolic execution with state interpolation (SI) has been shown to be effective for verifying sequential programs. In SI [15, 19, 14], a node at program point ℓ in the reachability tree can be pruned, if its context is subsumed by the interpolant computed earlier for the same program point ℓ. Therefore, even in the best case scenario, the number of states explored by an SI method must still

be at least the number of all *distinct* program points[1]. However, in the setting of concurrent programs, exploring each distinct global program point[2] once might already be considered prohibitive. In short, symbolic execution with SI *alone* is not efficient enough for the verification of concurrent programs.

Recent work (e.g., [27]) has shown the usefulness of going *stateful* in implementing a POR method. It directly follows that SI can help to yield even better performance. In order to implement an efficient stateful algorithm, we are required to come up with an abstraction for each (concrete or symbolic) state. Unsurprisingly, SI often offers us good abstractions.

The above suggests that POR and SI can be very much *complementary* to each other. In this paper, we propose a general framework employing *symbolic execution* in the exploration of the state space, while both POR and SI are exploited for pruning. SI and POR are combined synergistically as the concept of interpolation. Interpolation is essentially a form of learning where the completed search of a *safe* subtree is then formulated as a recipe, ideally a *succinct* formula, for future pruning. The key distinction of our interpolation framework is that each recipe discovered by a node is *forced* to be conveyed back to its ancestors, which gives rise to pruning of larger subtrees.

In summary, we address the challenge: "combining classic POR methods with symbolic technique has proven to be difficult" [16]. More specifically, we propose an algorithm schema to combine *synergistically* POR with state interpolation so that the sum is more than its parts. However, we first need to formalize POR wrt. a symbolic search framework with abstraction in such a way that: (1) POR can be *property driven* and (2) POR, or more precisely, the concept of persistent set, can be applicable for a set of states (rather than an individual state). While the main contribution is a theoretical framework (for space reason, we omit the proofs of the two theorems, they can be found at [4]), we also indicate a potential for the development of advanced implementations.

2 Related Work

Partial Order Reduction (POR) is a well-investigated technique in model checking of concurrent systems. Some notable early works are [23, 10]. Later refinements of POR, Dynamic [9] and Cartesian [12] POR (DPOR and CPOR respectively) improve traditional POR techniques by detecting collisions on-the-fly. These methods can, in general, achieve better reduction due to the more accurate detection of independent transitions.

One important weakness of traditional POR is that it is *insensitive* wrt. a target safety property. In contrast, recent works have shown that property-aware reduction can be achieved by symbolic methods using a general-purpose SAT/SMT

[1] Whereas POR-related methods do not suffer from this. Here we assume that the input concurrent program has already been preprocessed (e.g., by static slicing to remove irrelevant transitions, or by static block encodings) to reduce the size of the transition system for each process.

[2] The number of global points is the product of the numbers of local program points in all processes.

solver [26, 16, 24, 6]. Verification is often encoded as a formula which is *satisfiable* `iff` there exists an interleaving execution of the programs that violates the property. Reductions happen inside the SAT solver through the addition of learned clauses derived by conflict analysis [20]. This type of reduction is somewhat similar to what we call *state interpolation*.

The most relevant related work is [16], which is the first to consider enhancing POR with property driven pruning, via the use of an SMT solver. Subsequently, there was a follow-up work [24]. In [16], they began with an SMT encoding of the underlying transition system, and then they enhance this encoding with a concept of "monotonicity". The effect of this is that traces can be grouped into equivalence classes, and in each class, all traces which are *not monotonic* will be considered as *unsatisfiable* by the SMT solver. The idea of course is that such traces are in fact redundant. This work has demonstrated some promising results as most concurrency bugs in real applications have been found to be *shallow*. We note that [16] incidentally enjoyed some (weak) form of SI pruning, due to the similarity between conflict clause learning and state interpolation. However, there the synergy between POR and SMT is *unclear*. We later demonstrate in Sec. 7 that such synergy in [16] is indeed relatively poor.

There is a fundamental problem with scalability in [16], as mentioned in the follow-up work [24]: "It will not scale to the entire concurrent program" if we encode the whole search space as a single formula and submit it to an SMT solver.

Let us first compare [16] with our work. Essentially, the difference is twofold. First, in this paper, the theory for partial order reduction is *property driven*. In contrast, the monotonicity reduction of [16] is not. We specifically exemplify the power of property driven POR in the later sections. Second, the encoding in [16] is processed by a *black-box* SMT solver. Thus important algorithmic refinements are not possible. Some examples:

• There are different options in implementing SI. Specifically in this paper, we employ "precondition" computations. Using black-box solver, one has to rely on its fixed interpolation methods.

• Our approach is *lazy* in a sense that our solver is only required to consider *one* symbolic path at a time; in [16] it is not the case. This matters most when the program is unsafe and finding counter-examples is relatively easy (there are many traces which violate the safety).

• In having a symbolic execution framework, one can direct the search process. This is useful since the order in which state interpolants are generated does give rise to different reductions. Of course, such manipulation of the search process is hard, if not impossible, when using a black-box solver.

In order to remedy the scalability issue of [16], the work [24] adapted it to the setting of program testing. In particular, [24] proposed a concurrent trace program (CTP) framework which employs both concrete execution and symbolic solving to strike balance between efficiency and scalability of an SMT-based method. However, when the input program is *safe*, i.e., absence of bugs, [24] in general suffers from the same scalability issue as in [16].

We remark that, the new direction of [24], in avoiding the blow-up of the SMT solver, was in fact preceded by the work on under-approximation widening (UW) [11]. As with CTP, UW models a subset, which will be incrementally enlarged, of all the possible interleavings as an SMT formula and submits it to an SMT solver. In UW the scheduling decisions are also encoded as constraints, so that the *unsatisfiable core* returned by the solver can then be used to further the search in probably a useful direction. This is the major contribution of UW. However, an important point is that this furthering of the search is a *repeated* call to the solver, this time with a weaker formula; which means that the problem at hand is now larger, having more traces to consider. On this repeated call, the work done for the original call is thus *duplicated*.

At first glance, it seems attractive and simple to encode the problem compactly as a set of constraints and delegate the search process to a general-purpose SMT solver. However, there are some fundamental disadvantages, and these arise mainly because it is thus hard to exploit the semantics of the program to direct the search inside the solver. This is fact evidenced in the works mentioned above.

We believe, however, the foremost disadvantage of using a general-purpose solver lies in the encoding of process interleavings. For instance, even when a concurrent program has only *one* feasible execution trace, the encoding formula being fed to the solver is still of enormous size and can easily choke up the solver. More importantly, different from safety verification of sequential programs, the encoding of interleavings (e.g., [16] uses the variable *sel* to model which process is selected for executing) often hampers the normal derivations of succinct conflict clauses by means of resolution in modern SMT solvers. We empirically demonstrate the inefficiency of such approach in Sec. 7.

There have been other recent approaches addressing safety verification of concurrent programs. To name a few: [13] is based on CEGAR technology, [1] is based on symbolic execution, while [21, 22] are based on SMT. However, they digress from the POR family since they do not deal with the concept of swapping transitions.

Finally, we mention our previous work [5] where symmetric transformations of state interpolants are used to enhance symmetry reduction.While [5] relies on the fact that processes are defined parametrically and are roughly similar, this paper does not employ such assumptions.

3 Background

We consider a concurrent system composed of a finite number of threads or processes performing atomic operations on shared variables. Let P_i $(1 \leq i \leq n)$ be a process with the set $trans_i$ of transitions. Assume that $trans_i$ contains no cycles. Even though loops are important in concurrent systems, for simplicity, we ignore them (because loop-free programs are sufficient to exhibit our main contributions and routine techniques, e.g., as from [14], can always be employed to handle loops).

We also assume all processes have disjoint sets of transitions. Let $\mathcal{T} = \cup_{i=1}^{n} trans_i$ be the set of all transitions. Let V_i be the set of local variables

of process P_i, and V_{shared} the set of shared variables of the given concurrent program. Let $pc_i \in V_i$ be a special variable representing the process program counter, and the tuple $\langle pc_1, pc_2 \cdots, pc_n \rangle$ represent the global program point. Let *SymStates* be the set of all global symbolic states of the given program where $s_0 \in SymStates$ is the initial state. A state $s \in SymStates$ comprises two parts: its *global program point* ℓ, also denoted by $\texttt{pc}(s)$, which is a tuple of local program counters, and its *symbolic constraints* $[\![s]\!]$ over the program variables. In other words, we denote a state s by $\langle \texttt{pc}(s), [\![s]\!] \rangle$.

We consider the *transitions* of states induced by the program. Following [10], we only pay attention to *visible* transitions. A (visible) transition $t^{\{i\}}$ pertains to some process P_i. It transfers process P_i from control location ℓ_1 to ℓ_2. In general, the application of $t^{\{i\}}$ is guarded by some condition `cond` (`cond` might be just `true`). At some state $s \in SymStates$, when the i^{th} component of $\texttt{pc}(s)$, namely $\texttt{pc}(s)[i]$, equals ℓ_1, we say that $t^{\{i\}}$ is *schedulable*[3] at s. And when s satisfies the guard `cond`, denoted by $s \models \texttt{cond}$, we say that $t^{\{i\}}$ is *enabled* at s. For each state s, let $Schedulable(s)$ and $Enabled(s)$ denote the set of transitions which respectively are schedulable at s and enabled at s. A state s, where $Schedulable(s) = \emptyset$, is called a *terminal state*.

Let $s \xrightarrow{t} s'$ denote transition step from s to s' via transition t. This step is possible only if t is *schedulable* at s. We assume that the effect of applying an enabled transition t on a state s to arrive at state s' is well-understood. In our symbolic execution framework, executing a schedulable but not enabled transition results in an *infeasible* state. A state s is called *infeasible* if $[\![s]\!]$ is unsatisfiable. For technical reasons needed below, we shall allow schedulable transitions emanating from an infeasible state; it follows that the destination state must also be *infeasible*.

For a sequence of transitions w (i.e., $w \in \mathcal{T}^*$), $Rng(w)$ denotes the set of transitions that appear in w. Also let $\mathcal{T}_\ell$ denote the set of all transitions which are schedulable somewhere after global program point ℓ. We note here that the schedulability of a transition at some state s only depends on the program point component of s, namely $\texttt{pc}(s)$. It does not depend on the constraint component of s, namely $[\![s]\!]$. Given $t_1, t_2 \in \mathcal{T}$ we say t_1 can *de-schedule* t_2 `iff` there exists a state s such that both t_1, t_2 are schedulable at s but t_2 is not schedulable after the execution of t_1 from s.

Following the above, $s_1 \xRightarrow{t_1 \cdots t_m} s_{m+1}$ denotes a sequence of state transitions, and we say that s_{m+1} is reachable from s_1. We call $s_1 \xrightarrow{t_1} s_2 \xrightarrow{t_2} \cdots \xrightarrow{t_m} s_{m+1}$ a *feasible* derivation from state s_1, `iff` $\forall\ 1 \leq i \leq m \bullet t_i$ is enabled at s_i. As mentioned earlier, an *infeasible* derivation results in an *infeasible state* (an infeasible state is still aware of its global program point). An infeasible state satisfies any safety property.

We define a *complete execution* trace, or simply trace, ρ as a sequence of transitions such that it is a derivation from s_0 and $s_0 \xRightarrow{\rho} s_f$ and s_f is a

[3] This concept is not standard in traditional POR, we need it here since we are dealing with symbolic search.

terminal state. A trace is infeasible if it is an infeasible derivation from s_0. If a trace is infeasible, then at some point, it takes a transition which is schedulable but is not enabled. From thereon, the subsequent states are infeasible states.

We say a given concurrent program is *safe* wrt. a safety property ψ if $\forall s \in$ *SymStates* $\bullet$ if s *is reachable from the initial state* s_0 then $s \models \psi$. A trace ρ is *safe* wrt. ψ, denoted as $\rho \models \psi$, if all its states satisfy ψ.

Partial Order Reduction (POR) vs. State-based Interpolation (SI)

We assume the readers are familiar with the traditional concept of POR. Regarding state-based interpolation, we follow the approach of [15, 5]. Here our symbolic execution is depicted as a tree rooted at the initial state s_0 and for each state s_i therein, the descendants are just the states obtainable by extending s_i with a feasible transition.

Definition 1 (Safe Root). *Given a transition system and an initial state s_0, let s be a feasible state reachable from s_0. We say s is a* safe root *wrt. a safety property ψ, denoted $\bigtriangleup_\psi(s)$,* `iff` *all states s' reachable from s are safe wrt. ψ.*

Definition 2 (State Coverage). *Given a transition system and an initial state s_0 and s_i and s_j are two symbolic states such that (1) s_i and s_j are reachable from s_0 and (2) s_i and s_j share the same program point ℓ, we say s_i* covers *s_j wrt. a safety property ψ, denoted by $s_i \succeq_\psi s_j$,* `iff` *$\bigtriangleup_\psi(s_i)$ implies $\bigtriangleup_\psi(s_j)$.*

The impact of state coverage relation is that if (1) s_i covers s_j, and (2) the subtree rooted at s_i has been traversed and proved to be safe, then the traversal of subtree rooted at s_j can be avoided. In other words, we gain performance by *pruning* the subtree at s_j. Obviously, if s_i naturally subsumes s_j, i.e., $s_j \models s_i$, then state coverage is trivially achieved. In practice, however, this scenario does not happen often enough.

Definition 3 (Sound Interpolant). *Given a transition system and an initial state s_0, given a safety property ψ and program point ℓ, we say a formula $\overline{\Psi}$ is a* sound interpolant *for ℓ, denoted by $\mathsf{SI}(\ell, \psi)$, if for all state $s \equiv \langle \ell, \cdot \rangle$ reachable from s_0, $s \models \overline{\Psi}$ implies that s is a safe root.*

What we want now is to generate a formula $\overline{\Psi}$ (called *interpolant*), which still preserves the safety of all states reachable from s_i, but is weaker (more general) than the original formula s_i. In other words, we should have $s_i \models \mathsf{SI}(\ell, \psi)$. We assume that this condition is always ensured by any implementation of state-based interpolation. The main purpose of using $\overline{\Psi}$ rather than the original formula associated to the symbolic state s_i is to increase the likelihood of subsumption. That is, the likelihood of having $s_j \models \overline{\Psi}$ is expected to be much higher than the likelihood of having $s_j \models s_i$.

In fact, the perfect interpolant should be the weakest precondition [8] computed for program point ℓ wrt. the transition system and the safety property ψ. We denote this weakest precondition as $\mathsf{wp}(\ell, \psi)$. Any subsequent state $s_j \equiv \langle \ell, \cdot \rangle$ which has s_j stronger than this weakest precondition can be pruned.

However, in general, the weakest precondition is too computationally demanding. An interpolant for the state s_i is indeed a formula which approximates the weakest precondition at program point ℓ wrt. the transition system, i.e., $\overline{\Psi} \equiv \mathsf{SI}(\ell, \psi) \equiv \mathit{Intp}(s_i, \mathsf{wp}(\ell, \psi))$. A *good* interpolant is one which closely approximates the weakest precondition while can be computed efficiently.

The symbolic execution of a program can be augmented by annotating each program point with its corresponding interpolants such that the interpolants represent the sufficient conditions to preserve the unreachability of any unsafe state. Then, the *basic* notion of pruning with state interpolant can be defined as follows.

Definition 4 (Pruning with Interpolant). *Given a symbolic state $s \equiv \langle \ell, \cdot \rangle$ such that ℓ is annotated with some interpolant $\overline{\Psi}$, we say that s is* pruned *by the interpolant $\overline{\Psi}$ if s implies $\overline{\Psi}$ (i.e., $s \models \overline{\Psi}$).*

Now let us discuss the the effectiveness of POR and SI in pruning the search space with an example. For simplicity, we purposely make the example *concrete*, i.e., states are indeed concrete states.

EXAMPLE 1 *(Closely coupled processes)*: See Fig. 1. Program points are shown in angle brackets. Fig. 1(a) shows the control flow graphs of two processes. Process 1 increments x twice whereas process 2 doubles x twice. The transitions associated with such actions and the safety property are depicted in the figure. POR requires a full search tree while Fig. 1(b) shows the search space explored by SI. Interpolants are in curly brackets. Bold circles denote pruned/subsumed states.

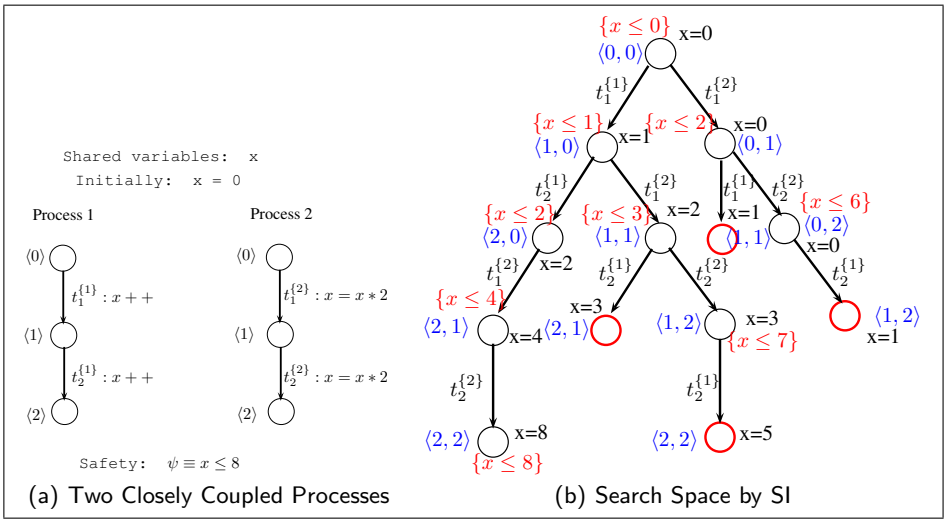

Fig. 1: Application of SI on 2 Closely Coupled Processes

Let us first attempt this example using POR. It is clear that $t_1^{\{1\}}$ is *dependent* with both $t_1^{\{2\}}$ and $t_2^{\{2\}}$. Also $t_2^{\{1\}}$ is dependent with both $t_1^{\{2\}}$ and $t_2^{\{2\}}$. Indeed, each of all the 6 execution traces in the search tree ends at a different concrete state. As classic POR methods use the concept of *trace equivalence* for pruning,

no interleaving is avoided: those methods will enumerate the full search tree of 19 states (for space reason, we omit it here).

Revisit the example using SI, where we use the *weakest preconditions* [8] as the state interpolants: the interpolant for a state is computed as the weakest precondition to ensure that the state itself as well as all of its descendants are safe (see Fig. 1(b)). We in fact achieve the best case scenario with it: whenever we come to a program point which has been examined before, subsumption happens. The number of non-subsumed states is still of order $O(k^2)$ (where $k = 3$ in this particular example), assuming that we generalize the number of local program points for each process to $O(k)$. Fig. 1(b) shows 9 non-subsumed states and 4 subsumed states.

In summary, the above example shows that SI might outperform POR when the component processes are closely coupled. However, one can easily devise an example where the component processes do not interfere with each other at all. Under such condition POR will require only one trace to prove safety, while SI is still (lower) bounded by the total number of global program points. In this paper, we contribute by proposing a framework to combine POR and SI *synergistically*.

4 Property Driven POR (PDPOR)

"Combining classic POR methods with symbolic algorithms has been proven to be difficult" [16]. The fundamental reason is that the concepts of (Mazurkiewicz) equivalence and transition independence, which drive all POR techniques, rely on the equivalence of two concrete states. However, in symbolic traversal, we rarely encounter two equivalent symbolic states.

We now make the following definition which is crucial for the concept of pruning and will be used throughout this paper.

Definition 5 (Trace Coverage). *Let ρ_1, ρ_2 be two traces of a concurrent program. We say ρ_1 covers ρ_2 wrt. a safety property ψ, denoted as $\rho_1 \sqsupseteq_\psi \rho_2$,* `iff` *$\rho_1 \models \psi \rightarrow \rho_2 \models \psi$.*

Instead of using the concept of trace equivalence, from now on, we only make use of the concept of trace coverage. The concept of trace coverage is definitely weaker than the concept of Mazurkiewicz equivalence. In fact, if ρ_1 and ρ_2 are (Mazurkiewicz) equivalent then $\forall \psi \bullet \rho_1 \sqsupseteq_\psi \rho_2 \wedge \rho_2 \sqsupseteq_\psi \rho_1$. Now we will define a new and *weaker* concept which therefore generalizes the concept of transition independence.

Definition 6 (Semi-Commutative After A State). *For a given concurrent program, a safety property ψ, and a derivation $s_0 \stackrel{\theta}{\Longrightarrow} s$, for all $t_1, t_2 \in \mathcal{T}$ which cannot de-schedule each other, we say t_1* semi-commutes *with t_2* after state *s wrt. $\sqsupseteq_\psi$, denoted by $\langle s, t_1 \uparrow t_2, \psi \rangle$,* `iff` *for all $w_1, w_2 \in \mathcal{T}^*$ such that $\theta w_1 t_1 t_2 w_2$ and $\theta w_1 t_2 t_1 w_2$ are execution traces of the program, then we have $\theta w_1 t_1 t_2 w_2 \sqsupseteq_\psi \theta w_1 t_2 t_1 w_2$.*

From the definition, $Rng(\theta)$, $Rng(w_1)$, and $Rng(w_2)$ are pairwise disjoint. Importantly, if s is at program point ℓ, we have $Rng(w_1) \cup Rng(w_2) \subseteq \mathcal{T}_\ell \backslash \{t_1, t_2\}$. We observe that wrt. some ψ, if all important events, those have to do with the safety of the system, have already happened in the prefix θ, the "semi-commutative" relation is trivially satisfied. On the other hand, the remaining transitions might still interfere with each other (but not the safety), and do not satisfy the independent relation.

The concept of "semi-commutative" is obviously weaker than the concept of independence. If t_1 and t_2 are independent, it follows that $\forall \psi\ \forall s \bullet \langle s, t_1\ \uparrow\ t_2, \psi\rangle \wedge \langle s, t_2\ \uparrow\ t_1, \psi\rangle$. Also note that, in contrast to the relation of transition independence, but similar to the concept of *weak stubborn* [23], the "semi-commutative" relation is *not symmetric*.

We now introduce a new definition for *persistent set*.

Definition 7 (Persistent Set Of A State). *A set $T \subseteq \mathcal{T}$ of transitions enabled in a state $s \in$* SymStates *is* persistent in *s wrt. a property ψ* `iff`*, for all derivations $s \xrightarrow{t_1} s_1 \xrightarrow{t_2} s_2 \ldots \xrightarrow{t_{m-1}} s_{m-1} \xrightarrow{t_m} s_m$ including only transitions $t_i \in \mathcal{T}$ and $t_i \notin T, 1 \leq i \leq m$, each transition in T* semi-commutes *with t_i after s wrt. $\sqsupseteq_\psi$.*

With the new definition of persistent set, we now can proceed with the normal *selective search* as described in classic POR techniques. In the algorithm presented in Fig. 2, we perform depth first search (DFS). For each state, computing a persistent set from the "semi-commutative" relation is similar to computing the classical persistent set under the transition independence relation. The algorithms for this task can be found at [10].

```
Safety property ψ and initial state s0
⟨1⟩ Initially : Explore(s0)
function Explore(s)
⟨2⟩ if s ⊭ ψ Report Error and TERMINATE
⟨3⟩ T := Persistent_Set(s)
⟨4⟩ foreach t in T do
⟨5⟩     s --t--> s'        /* Execute t */
⟨6⟩     Explore(s')
end function
```

Fig. 2: New Selective Search Algorithm

Theorem 1. *The selective search algorithm in Fig. 2 is* sound. □

In preparing for POR and SI to work together, we now further modify the concept of persistent set so that it applies for a set of states sharing the same program point. The previous definitions apply for a specific state only.

Definition 8 (Semi-Commutative After A Program Point). *For a given concurrent program, a safety property ψ, and $t_1, t_2 \in \mathcal{T}$, we say t_1* semi-commutes *with t_2* after program point *ℓ wrt. $\sqsupseteq_\psi$ and ϕ, denoted as $\langle \ell, \phi, t_1 \uparrow t_2, \psi\rangle$,* `iff` *for all state $s \equiv \langle \ell, \cdot\rangle$ reachable from the initial state s_0, if $s \models \phi$ then t_1 semi-commutes with t_2 after state s wrt. $\sqsupseteq_\psi$.*

Definition 9 (Persistent Set Of A Program Point). *A set $T \subseteq \mathcal{T}$ of transitions* schedulable *at program point ℓ is* persistent at *ℓ under the interpolant $\overline{\Psi}$ wrt. a property ψ* `iff`*, for all state $s \equiv \langle \ell, \cdot\rangle$ reachable from the initial state s_0,*

if $s \models \overline{\Psi}$ *then for all derivations* $s \xrightarrow{t_1} s_1 \xrightarrow{t_2} s_2 \ldots \xrightarrow{t_{m-1}} s_{m-1} \xrightarrow{t_m} s_m$ *including only transitions* $t_i \in \mathcal{T}$ *and* $t_i \notin T, 1 \leq i \leq m$*, each transition in* T *semi-commutes with* t_i *after state* s *wrt.* $\sqsupseteq_\psi$*.*

Assume that $T = \{tp_1, tp_2, \cdots tp_k\}$. The interpolant $\overline{\Psi}$ can now be computed as $\overline{\Psi} = \bigwedge \phi_{ji}$ for $1 \leq j \leq k, 1 \leq i \leq m$ such that $\langle \ell, \phi_{ji}, tp_j \uparrow t_i, \psi \rangle$.

For each program point, it is possible to have different persistent sets associated with different interpolants. In general, a state which satisfies a stronger interpolant will have a smaller persistent set, therefore, it enjoys more pruning.

5 Synergy of PDPOR and SI

We now show our combined framework. We assume for each program point, a persistent set and its associated interpolant are computed statically, i.e., by separate analyses. In other words, when we are at a program point, we can right away make use of the information about its persistent set.

The algorithm is in Fig. 3. The function Explore has input s_0 and assumes the safety property at hand is ψ. It naturally performs a depth first search of the state space.

```
Assume safety property ψ and initial state s0
⟨1⟩ Initially : Explore(s0)
function Explore(s)
    Let s be ⟨ℓ, ·⟩
⟨2⟩ if (memoed(s, Ψ̄)) return Ψ̄
⟨3⟩ if (s ⊭ ψ) Report Error and TERMINATE
⟨4⟩ Ψ̄ := ψ
⟨5⟩ ⟨T, Ψ̄_trace⟩ := Persistent_Set(ℓ)
⟨6⟩ if (s ⊨ Ψ̄_trace)
⟨7⟩     Ts := T
⟨8⟩     Ψ̄ := Ψ̄ ∧ Ψ̄_trace
⟨9⟩ else Ts := Schedulable(s)
⟨10⟩ foreach t in (Ts \ Enabled(s)) do
⟨11⟩         Ψ̄ := Ψ̄ ∧ pre(t, false)
⟨12⟩ foreach t in (Ts ∩ Enabled(s)) do
⟨13⟩     s --t--> s'              /* Execute t */
⟨14⟩     Ψ̄' := Explore(s')
⟨15⟩     Ψ̄ := Ψ̄ ∧ pre(t, Ψ̄')
⟨16⟩ memo and return (Ψ̄)
end function
```

Fig. 3: A Framework for POR and SI (DFS)

Two Base Cases: The function Explore handles two base cases. One is when the current state is subsumed by some computed (and memoed) interpolant $\overline{\Psi}$. No further exploration is needed, and $\overline{\Psi}$ is returned as the interpolant (line 2). The second base case is when the current state is found to be *unsafe* (line 3).

Combining Interpolants: We make use of the (static) persistent set T computed for the current program point. We comment further on this in the next section.

The set of transitions to be considered is denoted by Ts. When the current state implies the interpolant $\overline{\Psi}_{trace}$ associated with T, we need to consider only those transitions in T. Otherwise, we need to consider all the schedulable transitions. Note that when the persistent set T is employed, the interpolant $\overline{\Psi}_{trace}$ must contribute to the combined interpolant of the current state (line 8). Disabled transitions at the current state will strengthen the interpolant as in line 11. Finally, we recursively follow those transitions which are enabled at the current state. The interpolant of each child

state contributes to the interpolant of the current state as in line 15. In our framework, interpolants are propagated back using the precondition operation pre, where $\mathsf{pre}(t, \phi)$ denotes a *safe approximation* of the weakest precondition wrt. the transition t and the postcondition ϕ [8].

Theorem 2. *The algorithm in Fig. 3 is* sound. □

6 Implementation of PDPOR

We now elaborate on the remaining task: how to estimate the semi-commutative relation. Similar to the formalism of traditional POR, our formalism is of paramount importance for the semantic use as well as to construct the formal proof of correctness (see [4]). In practice, however, we have to come up with sufficient conditions to efficiently implement the concepts. In this paper, we estimate the semi-commutative relation in two steps:

1. We first employ *any* traditional POR method and first estimate the "semi-commutative" relation as the traditional independence relation (then the corresponding condition ϕ is just *true*). This is possible because the proposed concepts are *strictly weaker* than the corresponding concepts used in traditional POR methods.
2. We then identify and exploit a number of patterns under which we can statically derive and prove the semi-commutative relation between transitions. In fact, these simple patterns suffice to deal with a number of important real-life applications.

In the rest of this section, we outline three common classes of problems, from which the semi-commutative relation between transitions can be easily identified and proved, i.e., our step 2 becomes applicable.

Resource Usage of Concurrent Programs: Programs make use of limited resource (such as time, memory, bandwidth). Validation of resource usage in sequential setting is already a hard problem. It is obviously more challenging in the setting of concurrent programs due to process interleavings.

Here we model this class of problems by using a resource variable r. Initially, r is *zero*. Each process can increment or decrement variable r by some concrete value (e.g., memory allocation or deallocation respectively). A process can also double the value r (e.g., the whole memory is duplicated). However, the resource variable r cannot be used in the guard condition of any transition, i.e., we cannot model the behavior of a typical garbage collector. The property to be verified is that, "at all times, r is (upper-) bounded by some constant".

Proposition 1. *Let* r *be a resource variable of a concurrent program, and assume the safety property at hand is* $\psi \equiv \mathsf{r} \leq C$*, where* C *is a constant. For all transitions (assignment operations only)* $t_1 : \mathsf{r} = \mathsf{r} + c_1$*,* $t_2 : \mathsf{r} = \mathsf{r} * 2$*,* $t_3 : \mathsf{r} = \mathsf{r} - c_2$ *where* $c_1, c_2 > 0$*, we have for all program point* ℓ*:*

$\langle \ell, true, t_1 \uparrow t_2, \psi \rangle \wedge \langle \ell, true, t_1 \uparrow t_3, \psi \rangle \wedge \langle \ell, true, t_2 \uparrow t_3, \psi \rangle$ □

Informally, other than common mathematical facts such as additions can commute and so do multiplications and subtractions, we also deduce that additions can semi-commute with both multiplications and subtractions while multiplications can semi-commute with subtractions. This Proposition can be proved by using basic laws of algebra.

EXAMPLE 2 : Let us refer back to the example of two closely coupled processes introduced in Sec. 3, but now under the assumption that x is the resource variable of interest. Using the semi-commutative relation derived from Proposition 1, we need to explore only *one complete trace* to prove this safety.

We recall that, in contrast, POR (and DPOR)-only methods will enumerate the full execution tree which contains 19 states and 6 complete execution traces. Any technique which employs only the notion of Mazurkiewicz trace equivalence for pruning will have to consider all 6 complete traces (due to 6 different terminal states). SI alone can reduce the search space in this example, and requires to explore only 9 states and 4 subsumed states (as in Sec. 3).

Detection of Race Conditions: [25] proposed a property driven pruning algorithm to detect race conditions in multithreaded programs. This work has achieved more reduction in comparison with DPOR. The key observation is that, at a certain location (program point) ℓ, if their conservative "lockset analysis" shows that a search subspace is race-free, such search subspace can be pruned away. As we know, DPOR relies solely on the independence relation to prune redundant interleavings (if t_1, t_2 are independent, there is no need to flip their execution order). In [25], however, even when t_1, t_2 are dependent, we may skip the corresponding search space if flipping the order of t_1, t_2 does not affect the reachability of any race condition. In other words, [25] is indeed a (conservative) realization of our PDPOR, specifically targeted for detection of race conditions. Their mechanism to capture the such scenarios is by introducing a trace-based lockset analysis.

Ensuring Optimistic Concurrency: In the implementations of many concurrent protocols, *optimistic concurrency*, i.e., at least one process commits, is usually desirable. This can be modeled by introducing a flag variable which will be set when some process commits. The flag variable once set can not be unset. It is then easy to see that for all program point ℓ and transitions t_1, t_2, we have $\langle \ell, \mathsf{flag} = 1, t_1 \uparrow t_2, \psi \rangle$. Though simple, this observation will bring us more reduction compared to traditional POR methods.

7 Experiments

This section conveys two key messages. First, when trace-based and state-based methods are not effective individually, our combined framework still offers significant reduction. Second, property driven POR can be very effective, and applicable not only to academic programs, but also to programs used as benchmarks in the state-of-the-art.

We use a 3.2 GHz Intel processor and 2GB memory running Linux. Timeout is set at 10 minutes. In the tables, cells with '-' indicate timeout. We compare the

performance of Partial Order Reduction alone (POR), State Interpolation alone (SI), the synergy of Partial Order Reduction and State Interpolation (POR+SI), i.e., the semi-commutative relation is estimated using only step 1 presented in Sec. 6, and when applicable, the synergy of Property Driven Partial Order Reduction and State Interpolation (PDPOR+SI), i.e., the semi-commutative relation is estimated using both steps presented in Sec. 6. For the POR component, we use the implementation from [2].

Table 1 shows our first set of experiments. We start with parameterized versions of the *producer/consumer* example because its basic structure is extremely common. There are $2 * N$ producers and 1 consumer. Each producer will do its own non-interfered computation first, modeled by a transition which does not interfere with other processes. Then these producers will modify the shared variable x as follows: each of the first N producers increments x, while the other N producers double the value of x. On the other hand, the consumer consumes the value of x. The safety property is that the consumed value is no more than $N * 2^N$.

Table 1 clearly demonstrates the synergy benefits of POR and SI. POR+SI significantly outperforms both POR and SI. Note that this example can easily be translated to the resource usage problem, where our PDPOR requires only a *single* trace (and less than 0.01 second) in order to prove safety.

We next use the parameterized version of the *dining philosophers*. We chose this for two reasons. First, this is a classic example often used in concurrent algorithm design to illustrate synchronization issues and techniques for resolving them. Second, previous work [16] has used this to demonstrate benefits from combining POR and SMT.

Table 1: Synergy of POR and SI

	POR		SI		POR+SI	
Problem	States	T(s)	States	T(s)	States	T(s)
p/c-2	449	0.03	514	0.17	85	0.03
p/c-3	18745	2.73	6562	2.43	455	0.19
p/c-4	986418	586.00	76546	37.53	2313	1.07
p/c-5	–	–	–	–	11275	5.76
p/c-6	–	–	–	–	53261	34.50
p/c-7	–	–	–	–	245775	315.42
din-2a	22	0.01	21	0.01	21	0.01
din-3a	646	0.05	153	0.03	125	0.02
din-4a	155037	19.48	1001	0.17	647	0.09
din-5a	–	–	6113	1.01	4313	0.54
din-6a	–	–	35713	22.54	24201	4.16
din-7a	–	–	202369	215.63	133161	59.69
bak-2	48	0.03	38	0.03	31	0.02
bak-3	1003	1.85	264	0.42	227	0.35
bak-4	27582	145.78	1924	5.88	1678	4.95
bak-5	–	–	14235	73.69	12722	63.60

The first safety property used in [16], "it is not that all philosophers can eat simultaneously", is somewhat trivial. Therefore, here we verify a *tight* property, which is (a): "no more than *half* the philosophers can eat simultaneously". To demonstrate the power of symbolic execution, we verify this property *without* knowing the initial configurations of all the forks. Table 1, again, demonstrates the significant improvements of POR+SI over POR alone and SI alone. We note that the performance of our POR+SI algorithm is about 3 times faster than [16].

We additionally considered a second safety property as in [16], namely (b): "it is possible to reach a state in which all philosophers have eaten at least once". Our symbolic execution framework requires only a *single trace* (and less than 0.01 second) to prove this property in all instances, whereas [16] requires even more time compared to proving property (a). This illustrates the scalability issue of [16], which is representative for other techniques employing general-purpose SMT solver for symbolic pruning.

We also perform experiments on the "Bakery" algorithm. We note that, due to existence of infinite domain variables, model checking hardly can handle Bakery algorithm. The results are also shown in Table 1.

To further demonstrate the power our synergy framework over [16][4] as well as the power of our property driven POR, we experiment next on the *Sum-of-ids* program. Here, each process (of N processes) has one unique *id* and will increment a shared variable *sum* by this *id*. We prove that in the end this variable will be incremented by the sum of all the ids.

See Table 2, where we experiment with Z3 [7] (version 4.1.2) using the encodings presented in [16]. #C denotes the number of conflicts while #D denotes the number of decisions made by Z3. We can see that our synergy framework scale much better than [16] with Z3. Also, this example can also be translated to resource usage problem, our use of property-driven POR again requires *one* single trace to prove safety.

Table 2: Comparison with [16]

	[16] w. Z3			POR+SI		PDPOR+SI	
T(s)	#C	#D	T(s)	States	T(s)	States	T(s)
sum-6	1608	1795	0.08	193	0.05	7	0.01
sum-8	54512	59267	10.88	1025	0.27	9	0.01
sum-10	–	–	–	5121	1.52	11	0.01
sum-12	–	–	–	24577	8.80	13	0.01
sum-14	–	–	–	114689	67.7	15	0.01

Finally, to benchmark our framework with SMT-based methods, we select four *safe* programs from [6] where the experimented methods did not perform well. Those programs are micro_2, stack, circular_buffer, and stateful20. We note that safe programs allow fairer comparison between different approaches since to verify them we have to cover the whole search space. Table 3 shows the running time of SI alone and of the combined framework. For convenience, we also tabulate the *best* running time reported in [6] and C is the context switch bound used. We assume no context switch bound, hence the corresponding value in our framework is ∞.

Table 3: Experiments on [6]'s Programs

	[6]		SI		PDPOR+SI	
Problem	C	T(s)	States	T(s)	States	T(s)
micro_2	17	1095	20201	10.88	201	0.04
stack	12	225	529	0.26	529	0.26
circular_buffer	∞	477	29	0.03	29	0.03
stateful20	10	95	1681	1.13	41	0.01

We can see that even our SI alone significantly outperforms the techniques in [6]. We believe it is due to the inefficient encoding of process interleavings (mentioned in Sec. 2) as well as the following reasons. First, our method is *lazy*, which means that only a path is considered at a time: [6] itself demonstrates

[4] [16] is not publicly available. Therefore, it is not possible for us to make more comprehensive comparisons.

partially the usefulness of this. Second, but importantly, we are *eager* in discovering infeasible paths. The program circular_buffer, which has only one feasible complete execution trace, can be efficiently handled by our framework, but not SMT. This is one important advantage of our symbolic execution framework over SMT-based methods, as discussed in [19].

It is important to note that, PDPOR significantly improves the performance of SI wrt. programs micro_2 and stateful20. This further demonstrates the applicability of our proposed framework.

References

1. A. Albarghouthi et al. Abstract analysis of symbolic executions. In *CAV*, 2010.
2. P. Bokor et al. Supporting domain-specific state space reductions through local partial-order reduction. In *ASE*, 2011.
3. C. Cadar et al. Symbolic execution for software testing in practice: preliminary assessment. In *ICSE*, 2011.
4. D. H. Chu. *Interpolation Methods for Symbolic Execution*. Ph. D. Thesis, National University of Singapore, 2012.
5. D. H. Chu and J. Jaffar. A complete method for symmetry reduction in safety verification. In *CAV*, 2012.
6. L. Cordeiro and B. Fischer. Verifying multi-threaded software using smt-based context-bounded model checking. In *ICSE*, 2011.
7. L. De Moura and N. Bjørner. Z3: an efficient smt solver. In *TACAS*, 2008.
8. E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 1975.
9. C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *POPL*, 2005.
10. P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. Springer-Verlag, 1996.
11. O. Grumberg et al. Proof-guided underapproximation-widening for multi-process systems. In *POPL*, 2005.
12. G. Gueta et al. Cartesian partial-order reduction. In *SPIN*, 2007.
13. A. Gupta, C. Popeea, and A. Rybalchenko. Predicate abstraction and refinement for verifying multi-threaded programs. In *POPL*, 2011.
14. J. Jaffar, J.A. Navas, and A. Santosa. Unbounded Symbolic Execution for Program Verification. In *RV*, 2011.
15. J. Jaffar, A. Santosa, and R. Voicu. An interpolation method for clp traversal. In *CP*, 2009.
16. V. Kahlon, C. Wang, and A. Gupta. Monotonic partial order reduction: An optimal symbolic partial order reduction technique. In *CAV*, 2009.
17. J. C. King. Symbolic Execution and Program Testing. *Com. ACM*, 1976.
18. A. W. Mazurkiewicz. Trace theory. In *Advances in Petri Nets*, 1986.
19. K. McMillan. Lazy annotation for program testing and verification. In *CAV*, 2010.
20. J. P. M. Silva and K. A. Sakallah. Grasp-a new search algorithm for satisfiability. In *ICCAD*, 1996.
21. N. Sinha and C. Wang. Staged concurrent program analysis. In *FSE*, 2010.
22. N. Sinha and C. Wang. On interference abstractions. In *POPL*, 2011.
23. A. Valmari. Stubborn sets for reduced state space generation. In *Advances in Petri Nets*, 1991.
24. C. Wang et al. Symbolic pruning of concurrent program executions. In *FSE*, 2009.
25. C. Wang et al. Dynamic model checking with property driven pruning to detect race conditions. In *ATVA*, 2008.
26. C. Wang et al. Peephole partial order reduction. In *TACAS*, 2008.
27. Y. Yang et al. Efficient stateful dynamic partial order reduction. In *SPIN*, 2008.