A Framework for State and Trace Interpolation

Duc-Hiep Chu
National University of Singapore

Joxan Jaffar
National University of Singapore

Abstract—We address the problem of reasoning about interleavings in safety verification of concurrent processes. In the literature, there are two prominent techniques for pruning the search space: first, there is state-based interpolation where a collection of formulas can be generalized by taking into account the property to be verified. Second, there are trace-based methods, collectively known as “Partial Order Reduction”, which operate by weakening the concept of a trace by abstracting the total order in the trace into a partial order. We first contribute by further weakening the concept of Partial Order Reduction to Trace Interpolation, in order to adapt it for a symbolic execution framework with abstraction. The main contribution of this paper, however, is a framework that synergistically combines state and trace interpolation so that the sum is more than its parts.

I. INTRODUCTION

We address the search space explosion in the verification of concurrent programs due to the interleavings of transitions from different processes. In explicit-state model checking, a general approach is Partial Order Reduction (POR) [19], [5]. This exploits the equivalence of interleavings of ‘independent’ transitions, i.e. 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 [15] trace equivalence class of interleavings, if a representative has been checked, the remaining ones are regarded as redundant.

Symbolic execution [14] is a method for program reasoning that uses symbolic values as inputs instead of actual data, and it represents the values of program variables as symbolic expressions on the input symbolic values. A symbolic execution tree depicts all executed paths during the symbolic execution. A path condition is maintained for each path: it is a formula over the symbolic inputs by accumulating constraints which those inputs must satisfy in order for execution to follow that path. A path is infeasible if its path condition is unsatisfiable. Otherwise, the path is feasible.

The main challenge for symbolic execution is the exponential number of symbolic paths. The works [12], [17], [11] tackle successfully this fundamental problem by eliminating from the concrete model 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 (SI) in a similar spirit to that of conflict clause learning in SAT solvers.

Now SI has been shown to be effective. In SI [12], [11], a node at program point pp in the reachability tree can be pruned, if its context is subsumed by the interpolant computed earlier for the same program point pp. Therefore, even in the best case scenario, the number of states explored by a 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 [23] 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 good abstraction for each (concrete or symbolic) state. Unsurprisingly, SI often offers us good (and sometimes even the best) abstraction.

The above suggests that POR and SI can be very much complementary to each other. In this paper, we propose a general framework with explicit state space exploration, while both POR and SI are exploited for pruning. POR is realized by a notion of trace interpolation whose purpose is branch pruning; in contrast, SI is realized by the (standard) notion of state interpolation whose purpose is state pruning. These two concepts are combined synergistically as the concept of interpolation. Interpolation, in general, is essentially a form of learning where the completed search of a safe subtree is then formulated as a recipe for future pruning. The key idea is that each recipe discovered by a node will be conveyed back to its ancestors, which gives rise to pruning of larger subtrees. Another important distinction is that our method learns symbolically with respect to the safety property and the interleavings. Consequently, we achieve more reductions compared to the state-of-the-art.

In summary, we address the challenge: “combining classic POR methods with symbolic technique has proven to be difficult” [13]. More specifically, this paper makes two conceptual contributions:

- We generalize the concept of POR to the concept of trace interpolation. The purpose here is to formalize POR wrt. a traditional symbolic execution framework with abstraction in such a way that: (1) POR can be property dependent and (2) POR can be seamlessly used together with SI. Note that in a symbolic execution framework, we no longer can rely on the concept of (Mazurkiewicz) trace equivalence. Instead, this

\(^1\)Whereas POR-related methods do not suffer from this.
\(^2\)The number of global points is the product of the numbers of program points in each process.
A. Related Work

Partial Order Reduction (POR) is a well-investigated technique in model checking of concurrent systems. Some notable early works are [19], [5]. Later refinements of POR, Dynamic [4] and Cartesian [7] 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 solver. Verification is often encoded as a formula which is satisfiable if 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 [18]. We associate this type of reduction to state interpolation (SI).

Another related work on reducing interleavings is [8], but this is not POR based. It is based on CEGAR [9], [16] technology. It considers ‘rely-guarantee’ approach to obtain global reasoning by considering each process in isolation, and using ‘environment transitions’ to summarize the effect of executing other processes on the current process. These transitions can be abstracted so as to expose only a small amount of information of process interleaving. And following the CEGAR methodology, if this abstraction is too coarse to prove the target property, abstraction refinement is performed. This approach would remove non-interfering transitions from consideration. However, when transitions are closely relevant to the proof, it does not enjoy the advantages of POR because it does not explicitly deal with the concept of swapping transitions. Moreover, in the worst case, this approach will enumerate the whole global search tree for each process.

The most relevant related work to ours is [13] because it is the first to consider a combination of the POR and SMT approaches. Subsequently, there was a follow-up work [20].

In [13], 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 SAT 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 application have been found to be shallow.

However, there is a fundamental problem with scalability in [13], as mentioned in the follow-up work [20]: “It will not scale to the entire concurrent program” if we encode the whole search space as a single formula and submit it to a SAT/SMT solver.

Before describing [20], we compare [13] with our work in this paper. Essentially, the difference is twofold:

First, our concept of POR reduction is property dependent. In contrast, the monotonicity reduction of [13] is not property dependent. Consequently, if exploited properly, our trace interpolation will give better pruning than [13]. We specifically exemplify this below.

Second, the encoding in [13] is processed by a black-box SAT/SMT solver. Thus important algorithmic refinements are not possible. Some examples:

- we employ a “precondition” computation which gives rise to more pruning than a general-purpose solver. In Figure 1(c) for example, we will show a formula (e.g. \( x + y \leq 3 \)) that does not originally appear in the SMT encoding can be a better interpolant than what arises from the standard technique using “unsatisfiable cores” (which only involve the original formulas).
- Our approach is incremental in a sense that our solver is only required to consider one symbolic path at a time; in [13] it is not the case.
- In having a symbolic execution framework, one can direct the search process. This can be useful since the order in which interpolants are generated in the SI process does give rise to different reductions. Of course, such manipulation of the search process is not possible when using a black-box solver.

In order to remedy the scalability issue of [13], the work [20] proposed a concurrent trace program (CTP) framework which employs both concrete execution and symbolic solving to strike balance between efficiency and scalability of SAT-based method. The new direction of [20], in avoiding the blow-up of the SAT solver, was in fact preceded by the work on under-approximation widening (UW) [6]. As with CTP, UW models a subset, which will be incrementally enlarged, of all the possible interleavings as a SAT formula and submits it to a SAT solver. In UW the scheduling decisions are also encoded as constraints, so that the unsatisfiable core returned by 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.

We finally mention the recent paper [2] which considers some engineering aspects of UW and provides extensive evaluations of previous SAT-based techniques.

We conclude this subsection with a few observations about using a black-box solver. In general, it is 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 related works mentioned above.

The foremost disadvantage of a general-purpose solver is that its main strength is focused on its ability to determine unsatisfiability of a single conjunction of formulas (in our setting this means a single trace). Critically, it has to produce what we call an interpolant, a simpler formulation or “core” of the conjunction which demonstrates the unsatisfiability. This helps in pruning of other conjunctions which share the same core. Now, in a tree of interleavings, the interpolant we desire is, as in POR, a formula that encapsulates traces which are formed by having transitions in different order. Such an interpolant is not yet available (it is in fact one of the contributions of this paper).

A second problem with using a general-purpose solver is that successive calls to the solver usually correspond to a number of traces where a (strict) subset of them have already been resolved in previous calls. Thus, the search process is not incremental in two ways: (a) what has been done before might be redone again (b) and also importantly, what has been done before, cannot be learned to support the search in the future. We note that, making the theory solvers (in SMT) more incremental would not resolve this problem.

II. BACKGROUND AND DISCUSSIONS

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 $\text{trans}_i$ of transitions. For simplicity, assume that $\text{trans}_i$ contains no cycles. Even though loops are important in concurrent systems, we simply ignore them (because loop-free programs are sufficient to exhibit our main contributions).

We also assume all processes have disjoint sets of transitions. Let $T = \bigcup_{i=1}^N \text{trans}_i$ be the set of all transitions. Let $V_i$ be the set of local variables of process $P_i$, and $V_{\text{shared}}$ the set of shared variables of the given concurrent program. Let $\text{pc}_i \in V_i$ be a special variable representing the program process counter, and the tuple $\langle \text{pc}_1, \text{pc}_2, \ldots, \text{pc}_N \rangle$ represent the global program point. Let $\text{State}$ be the set of all global symbolic states of the given program where $s_0 \in \text{State}$ is the initial state. A state $s \in \text{State}$ comprises two parts: its program point $\text{pc}(s)$, which is a tuple of local program counters, and its symbolic constraints $\text{cons}(s)$ over the program variables. We denote a state $s$ by $\langle \text{pc}(s); \text{cons}(s) \rangle$.

We consider the transitions of states induced by the program. Following [5], we only pay attention to visible transitions. A (visible) transition $t$ pertains to some process $P_i$. It transfers process $P_j$ from control location $l_1$ to $l_2$. In general, the application of $t$ is guarded by some condition $\text{cond}$ (cond might be just $\text{true}$). At some state $s \in \text{State}$, when the $i^{th}$ component of $\text{pc}(s)$, namely $\text{pc}(s)[i]$, equals $1$, we say that $t$ is schedulable at $s$. And when the symbolic constraints $\text{cons}(s)$ satisfies the guard $\text{cond}$, denoted by $\text{cons}(s) \models \text{cond}$, we say that $t$ is enabled at $s$. For each state $s$, let $\text{Scheduled}(s)$ and $\text{Enabled}(s)$ denote the set of transitions which respectively are schedulable at $s$ and enabled at $s$. A state $s$, where $\text{Scheduled}(s) = \emptyset$, is called a terminating state.

Let $s \xrightarrow{t} s'$ denote transition step from $s$ to $s'$ via $t$. 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 a false state. A state $s$ is called false if its constraints $\text{cons}(s)$ are unsatisfiable. For technical reasons needed below, we shall allow schedulable transitions emanating from a false state; it follows that the destination state must also be false.

Example 1. Consider two processes $P_1, P_2$: $P_1$ simply awaits for $x = 0$, while $P_2$ increments $x$. So each has one transition to transfer (locally) from control location $(0)$ to $(1)$. Assume that initially $x = 0$, i.e., the initial state $s_0$ is $\langle (0, 0); x_0 = 0 \rangle$. Running $P_2$, first we have the transition from state $\langle (0, 0); x_0 = 0 \rangle$ to state $\langle (0, 1); x_1 = x_0 + 1 \land x_0 = 0 \rangle$. From here, we note that the transition from $P_1$ is now not enabled even though it is schedulable. If applied, it produces a false (and terminating) state $\langle (1, 1); x_1 = 0 \land x_1 = x_0 + 1 \land x_0 = 0 \rangle$. Note that $(x_1 = 0 \land x_1 = x_0 + 1 \land x_0 = 0) \equiv \text{false}$.

On the other hand, running $P_1$ first, we have the transition from $\langle (0, 0); x_0 = 0 \rangle$ to $\langle (1, 0); x_0 = 0 \rangle$. We may now have a subsequent transition step to $\langle (1, 1); x_0 = 0 \land x_1 = x_0 + 1 \rangle$ which is a non-false terminating state.

For simplicity, for each symbolic state $s$, we henceforth write the constraint component of $s$ in its intuitive normalized form, instead of using SSA form as in Example 1.

For a sequence of transitions $w$ (i.e. $w \in T^*$), $\text{Rng}(w)$ denotes the set of transitions that appear in $w$. Also let $T_{pp}$ denote the set of all transitions which is schedulable somewhere after program point $pp$. We note here that the schedulability of a transition at some state $s$ only depends on the program point component of $s$, namely $\text{pc}(s)$. It does not depend on the constraints component of a state.

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

We define a (complete execution) trace $\rho$ 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 terminating 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 false states.

We say the given concurrent system is safe wrt. a safety property $\psi$ if $\forall s \in \text{State} :$ if $s$ is reachable from the initial
A trace $\rho$ is safe wrt. $\varphi$, denoted as $\rho \models \varphi$, if all its states satisfy $\varphi$.

### A. Partial Order Reduction

POR methods exploit the fact that paths often differ only in execution order of non-interacting or “independent” transitions. This notion of independence between transitions can be formalized by the following definitions [5]:

**Definition 1** (Independence Relation). $\mathcal{R} \subseteq T \times T$ is an independence relation iff for each $\langle t_1, t_2 \rangle \in \mathcal{R}$ the following properties hold for all $s \in S$:

1. if $t_1$ is enabled in $s$ and $s \xrightarrow{t_1} s'$, then $t_2$ is enabled in $s$ iff $t_2$ is enabled in $s'$; and
2. if $t_1$ and $t_2$ are enabled in $s$, then there is a unique state $s''$ such that $s \xrightarrow{t_1} s''$ and $s \xrightarrow{t_2} s''$.

**Definition 2** (Equivalence). Two traces are (Mazurkiewicz) equivalent iff one trace can be transformed into another by repeatedly swapping adjacent independent transitions.

Traditional algorithms employ the concept of trace equivalence for pruning. They operate as classic state space searches except that, at each encountered state $s$, they compute the subset $T$ of the transitions enabled at $s$, and explore only the transitions in $T$. Intuitively, a subset $T$ of the set of transitions enabled in a state $s$ is called persistent in $s$ if whatever one does from $s$, while remaining outside of $T$, does not interact with $T$. Formally, we have the following:

**Definition 3** (Persistent). A set $T \subseteq T$ of transitions enabled in a state $s \in State$ is persistent in $s$ iff, for all feasible derivations $s \xrightarrow{t_1} s_1 \xrightarrow{t_2} \cdots \xrightarrow{t_{n-1}} s_{n-1} \xrightarrow{t_n} s_n$ including only transitions $t_i \in T$ and $t_i \notin T$, $1 \leq i \leq n$, $t_i$ is independent with all the transitions in $T$.

Now let us briefly illustrate the application of POR to two simple but contrasting examples.

**Example 2** (Loosely coupled processes). Figure 1(a) shows the control flow graphs of two processes. Process 1 increments $x$ twice whereas process 2 increments $y$ twice. The transitions associated with such actions and the safety property are depicted in the figure.

In Example 2, $t_{11}$ is independent with both $t_{21}$ and $t_{22}$. Also $t_{12}$ is independent with both $t_{21}$ and $t_{22}$. Thus for all the non-terminating states shown in Figure 1(b), the persistent set of that state contains only one transition. Therefore, only one trace is needed by POR.

**Example 3** (Closely coupled processes). See Figure 2(a). 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.

Let us attempt this example using POR. It is clear that $t_{11}$ is dependent with both $t_{21}$ and $t_{22}$. Also $t_{12}$ is dependent with both $t_{21}$ and $t_{22}$. Indeed, each of all the 6 execution traces in the search tree ends at a different concrete state. As classic

<table>
<thead>
<tr>
<th>Fig. 1 Two loosely coupled processes (a) POR performance (b) SI performance (c). Arrows without ending circle denote pruned transitions. Interpolants are (red) in curly brackets. Bold red circles denote pruned/subsumed states.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fig. 2 Two closely coupled processes (a) and SI performance (b). Interpolants are in curly brackets. Bold circles are pruned states.</td>
</tr>
</tbody>
</table>

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 (omitted for space reasons). In contrast, Figure 2(b) shows the search space explored by SI, comprising 9 states and 4 subsumed states.

### B. State Interpolation

State-based interpolation (SI) was first described in [12] for finite transition systems. The essence was to prune the search space of symbolic execution, informally described as
follows. Symbolic execution is usually depicted as a tree rooted at the initial state \( s_0 \) and for each state \( s_i \), the descendants are just the states obtainable by extending \( s_i \) with an enabled transition. Consider one particular feasible path: \( s_0 \xrightarrow{t_1} s_1 \xrightarrow{t_2} s_2 \cdots s_m \). The boundary between each transition can be considered a program point, characterizing a point in the reachability tree in terms of all the remaining possible transitions. Now, this particular path is safe wrt. a safety property \( \psi \) if for all \( i \), \( 0 \leq i \leq m \), we have \( s_i \models \psi \). A (state) interpolant at program point \( j \), \( 0 \leq j \leq m \) is simply a set of states \( S_j \) containing \( s_j \) such that for any state \( s' \in S_j \), \( s'_i \xrightarrow{t_{i+1}} s'_{i+1} \xrightarrow{t_{i+2}} \cdots s'_{j+2} \xrightarrow{t_{j+2}} s'_{m} \), it is also the case that for all \( i, j \leq i \leq m \), we have \( s'_i \models \psi \). This interpolant was constructed at point \( j \) due to the one path. Consider now all paths from \( s_0 \) and with prefix \( t_1, \cdots, t_{j-1} \). Compute each of their interpolants. Finally, the interpolant for the subtree (at \( s_j \)) of paths just considered is simply the intersection of all the individual interpolants. This notion of interpolant for a subtree provides a notion of subsumption. We can now prune a subtree in case its root is within the interpolant computed for a previously encountered subtree of the same program point.

Let us revisit Example 2 using SI. We in fact achieve the best case scenario with it: whenever we come to a program point which has been examined before, subsumption happens. Here we just use the weakest preconditions [3] as the state interpolants. The number of states explored is still of order \( a \) previously encountered subtree of the same program point. A subtree in case its root is within the interpolant computed for a point in the reachability tree in terms of all the remaining possible transitions. Now,\( a \) particular path is safe wrt. a safety property \( \psi \) if for all \( i \), \( 0 \leq i \leq m \), we have \( s_i \models \psi \). A (state) interpolant at program point \( j \), \( 0 \leq j \leq m \) is simply a set of states \( S_j \) containing \( s_j \) such that for any state \( s' \in S_j \), \( s'_i \xrightarrow{t_{i+1}} s'_{i+1} \xrightarrow{t_{i+2}} \cdots s'_{j+2} \xrightarrow{t_{j+2}} s'_{m} \), it is also the case that for all \( i, j \leq i \leq m \), we have \( s'_i \models \psi \). This interpolant was constructed at point \( j \) due to the one path. Consider now all paths from \( s_0 \) and with prefix \( t_1, \cdots, t_{j-1} \). Compute each of their interpolants. Finally, the interpolant for the subtree (at \( s_j \)) of paths just considered is simply the intersection of all the individual interpolants. This notion of interpolant for a subtree provides a notion of subsumption. We can now prune a subtree in case its root is within the interpolant computed for a previously encountered subtree of the same program point.

Let us revisit Example 2 using SI. We in fact achieve the best case scenario with it: whenever we come to a program point which has been examined before, subsumption happens. Here we just use the weakest preconditions [3] as the state interpolants. The number of states explored is still of order \( O(n^2) \) (where \( n = 3 \) in this particular example), assuming that we generalize the number of local program points for each process to \( O(n) \). The search tree (9 states and 4 subsumed states) is depicted as in Figure 1(c).

Now we reconsider Example 3 using SI. One possible search tree is depicted in Figure 2(b). Note the similarity between the tree in Figure 2(b) and the tree in Figure 1(c), even though Example 3 contains more interfering transitions, comparing to Example 2.

In summary, Examples 2 and 3 show that SI and POR each will perform better or worse than each other under certain conditions. It would then be a great contribution if one can combine them in one to exploit the synergy of both SI and POR. In fact, we achieve this with section VI.

III. STATE INTERPOLATION REVISITED

See Figure 3. From \( s_0 \), by following one particular feasible sequence \( \theta_1 \), we reach state \( s_i \) which is at global program point \( pp \). Let \( \mathcal{W}_{pp} \) denote the set of all possible suffix traces starting from program point \( pp \). For a sequence \( w \in \mathcal{W}_{pp} \), \( w \) may be a feasible or infeasible derivation from \( s_i \).

We assume that the subtree \( A \) rooted at \( s_i \) has been explored and proved safe wrt. property \( \psi \). The question now is: in subsequent (depth-first) traversal via \( \theta_2 \) we reach state \( s_j \) which is also at the program point \( pp \), can we avoid considering the subtree at \( s_j \), namely subtree \( A' \)? If so, we consider this scenario as state pruning. We now make the following definitions which are crucial for the concept of pruning and will be used throughout this paper.

**Definition 4** (Trace coverage). Let \( \rho_1, \rho_2 \) be two traces of a concurrent system. We say \( \rho_1 \) covers \( \rho_2 \) wrt. a property \( \psi \), denoted as \( \rho_1 \supseteq \psi \rho_2 \), iff \( \rho_1 \vdash \psi \rightarrow \rho_2 \vdash \psi \).

**Definition 5** (Safe Root). Let \( \theta_1 \) be a sequence of transitions s.t. \( s_0 \xrightarrow{\theta_1} s_i \), we say that \( s_i \) is a safe root wrt. a property \( \psi \), denoted \( \Delta_\psi(s_i) \), iff all states \( s'_i \) reachable from \( s_i \) are safe.

**Definition 6** (State Coverage). Let \( \theta_1 \) and \( \theta_2 \) be two sequences of transitions such that (1) \( s_0 \xrightarrow{\theta_1} s_i \), (2) \( s_0 \xrightarrow{\theta_2} s_j \), (3) \( s_i \) and \( s_j \) share the same program point \( pp \). We say that \( s_i \) covers \( s_j \) wrt. a property \( \psi \), denoted by \( s_i \models \psi \rightarrow s_j \models \psi \) and therefore the subtree rooted at \( s_j \) can be avoided (or pruned), iff \( \Delta_\psi(s_i) \) implies \( \Delta_\psi(s_j) \).

One way to determine whether \( s_i \) covers \( s_j \) is to make use of the trace coverage concept. We can conclude that \( s_i \) covers \( s_j \) if \( \forall w \in \mathcal{W}_{pp}, \theta_1 w \supseteq \psi \theta_2 w \). This seems elegant as it takes care of both feasible and infeasible traces. However, for some \( w \in \mathcal{W}_{pp} \), if \( \theta_1 w \) is infeasible, i.e. at some point a symbolic false state is reached, this trace will trivially satisfy \( \psi \) from that point on. It is then hard to ensure the safety of \( \theta_2 w \) without exploring the subtree rooted at \( s_j \). As a matter of fact, in practice, during the exploration of subtree rooted at \( s_i \), we also compute a state-interpolant of \( s_i \), denoted as \( \Delta_\psi(s_i) \), where \( \Delta_\psi(s_i, \psi) \) ensures that for all state \( s_j \) at program point \( pp \), if \( s_j \models \Delta_\psi(s_i, \psi) \) then \( \forall t \in \text{Scheduled}(s_i) \) (note that \( \text{Scheduled}(s_i) = \text{Scheduled}(s_j) \)) the two following conditions must be satisfied:

- if \( t \) was disabled at \( s_i \), it also must be disabled at \( s_j \)
- if \( t \) is enabled at \( s_i \) (by the above condition, \( t \) must be enabled at \( s_i \) too) and \( s_j \xrightarrow{t} s'_j \) and \( s_i \xrightarrow{t} s'_i \), then \( s'_i \) must cover \( s'_j \).

This observation enables us to compute the interpolants recursively. We will realize this idea for the state interpolation component of the synergy algorithm in section VI.

IV. PROPERTY DRIVEN POR

“Combining classic POR methods with symbolic algorithms has been proven to be difficult” [13]. 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.

It is even more difficult extending POR to be property driven. The difficulty arises from the fact that symbolic methods implicitly manipulate large sets of states as opposed to
states individually. Capturing and exploiting transitions which are dynamically independent with respect to a set of states is thus much harder.

Let us next discuss about the traditional concept of transition independence. Here we ignore the matter of enablement and disablement of transitions. In order for two transitions \( t_1 \) and \( t_2 \) to be independent, it is required that for all state \( s \) in the state space \( \text{State} \), there is a unique state \( s' \) such that \( s \xrightarrow{t_1,t_2} s' \) and \( s \xrightarrow{t_2,t_1} s' \). Indeed, this requirement for the uniqueness of \( s' \), which in general is not satisfied in symbolic setting, hinders the extension of POR to symbolic techniques.

\[ \text{Fig. 4 Branch Pruning} \]

See Figure 4. Assume that we have finished examining the subtree \( A \), resulting from taking transition \( t_1 \) at state \( s_i \). Now the question again is whether we can avoid those subtrees resulting from not taking \( t_1 \). Do we really need \( t_1 \) to be independent with those transitions appearing in the suffix subtree (in \( A \))? In fact, the question becomes whether \( t_1 \) can commute forward over each of such transitions so that it can blend in freely with those suffixes without affecting the safety. For example, we do not require \( t_2 \) to be able to commute forward over \( t_1 \) safely in order for the branch by taking transition \( t_2 \) to be pruned. Indeed, \( t_2 \) must appear somewhere in \( A \) (if not, we definitely cannot prune). In other words, in those traces, \( t_1 \) is always taken before \( t_2 \). Informally, the swapping is required to be one-way only.

Instead of using the concept of trace equivalence, from now on, we only make use of the concept of trace coverage. Such concept of trace coverage is definitely weaker than the concept of Mazurkiewicz equivalence. In fact, if \( \rho_1 \) and \( \rho_2 \) are (Mazurkiewicz) equivalent then (a) \( \rho_1 \) and \( \rho_2 \) have the same set of transitions (b) \( \forall \psi, \rho_1 \equiv_\psi \rho_2 \wedge \rho_2 \equiv_\psi \rho_1 \). Now we will define a new and weaker concept which therefore generalizes the concept of transition independence.

**Definition 7** (Semi-Commutative After A State). For all \( \theta, w_1, w_2 \in T^* \) and for all \( t_1, t_2 \in T \) and for all property \( \psi \) such that (1) \( s_0 \xrightarrow{\theta} s \) is a feasible derivation; (2) \( \theta w_1 t_1 t_2 w_2 \) and \( \theta w_1 t_2 t_1 w_2 \) both are execution traces of the program, we say \( t_1 \) can semi-commute with \( t_2 \) after state \( s \) wrt. \( \equiv_{\psi} \), denoted by \( \langle s, t_1 \uparrow t_2, \psi \rangle \) iff \( \varnothing \theta w_1 t_1 t_2 w_2 \equiv_{\psi} \theta w_1 t_2 t_1 w_2. \)

We note that in the above definition \( t_1 \) and \( t_2 \) cannot dischedule each other, i.e. they belong to different processes. Also from the definition, \( \text{Rng}(\theta), \text{Rng}(w_1), \) and \( \text{Rng}(w_2) \) are pairwise disjoint. Importantly, if \( s \) is at program point \( pp \), we have \( \text{Rng}(w_1) \subseteq \text{Rng}(w_2) \subseteq \langle pp \rangle \setminus \{t_1, t_2\}. \)

We observe that wrt. some \( \psi \), if all important events have already happened in the prefix \( \theta \), the ‘semi-commutative’ relation is trivially satisfied. Also, though two transitions \( t_1 \) and \( t_2 \) might interfere with each other and are both important to ensure \( \psi \), but all transitions which might interfere with them have already happened in the prefix \( \theta \), deciding whether \( t_1 \) can semi-commute with \( t_2 \) (or vice versa) is easy while it might give rise to significant pruning.

The concept of ‘semi-commutative’ is obviously weaker than that of transition independence. If \( t_1 \) and \( t_2 \) are independent, it implies that \( \forall \psi \forall s, \langle s, t_1 \uparrow t_2, \psi \rangle \wedge \langle s, t_2 \uparrow t_1, \psi \rangle \). Also note that, in contrast to the relation of transition independence, but similar to the concept of weak stubborn [19], the ‘semi-commutative’ relation is not symmetric.

We now introduce a new definition for persistent set.

**Definition 8** (Persistent Set Of A State). A set \( T \subseteq T \) of transitions enabled in a state \( s \in \text{State} \) is persistent in \( s \) wrt. a property \( \psi \) iff, for all feasible derivations \( s \xrightarrow{t_1,s_1 \cdots s_n} \) \( s_2 \cdots s_n \xrightarrow{t_n-1} s_{n-1} \xrightarrow{t_n} s_n \) including only transitions \( t_i \in T \) and \( t_i \notin T, 1 \leq i \leq n \), each transition in \( T \) can semi-commute with \( t_i \) after \( s \) wrt. \( \exists \psi \).

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 figure 5, we perform depth first search (DFS). As commonly known, for each state, computing a good persistent set from the ‘semi-commutative’ relation is not a trivial task. However, the task is similar to computing the classical persistent set under the transition independence relation. The algorithms for this task can be found elsewhere ([5]).

\[ \text{Fig. 5 Persistent-Set Selective Search (DFS). Assume safety property } \psi \text{ and initial state } s_0. \]

| (1) Initially : Explore\((s_0)\); |
| function Explore\((s)\) |
| (2) If \( s \not\in \psi \) Report Error |
| (3) \( T = \text{Persistent Set}(s) \); |
| (4) foreach \( t \in T \) do |
| (5) \( \triangleright s = \text{success}(s) \) after \( t \) /* Execute \( t \)*/ |
| (6) Explore\((t)\); |
| (7) endfor |
| end function |

**Lemma 1.** The algorithm in Figure 5 is sound.

**Proof Outline.** Assume that there exist some traces which violate the property \( \psi \) and are not examined by our selective search. Let denote the set of such traces as \( W_{\text{violated}} \). For each trace \( \rho = s_0 \xrightarrow{t_1} s_1 \xrightarrow{t_2} \cdots \xrightarrow{t_n} s_n \), \( \rho \in W_{\text{violated}} \), let \( \text{first}(\rho) \) denote the smallest index \( i \) such that \( t_i \) is not in the persistent set of \( s_{i-1} \). Without loss of generality, assume \( \rho_{\text{max}} = s_0 \xrightarrow{t_1} s_1 \xrightarrow{t_2} s_2 \cdots \xrightarrow{t_n} s_n \) having the maximum \( \text{first}(\rho) \). Let \( \text{first}(\rho_{\text{max}}) = i \). As the 'commuter'
and 'commutee' cannot dis-schedule each other, in the set \{t_{i+1} \cdots t_n\} there must be a transition which belongs to the persistent set of s_{i-1} (otherwise, the must exist some transition that belongs to the persistent set of s_{i-1} which is schedulable at s_n. Therefore s_n is not a terminating state). Let j be the smallest index such that t_j belongs to the persistent set of s_{i-1}. By definition, wrt \ci \psi and after s_{i-1}, t_j can semi-commute with t_{i-1}, t_{i+1}, \cdots t_{j-1}. Also due to the definition of the 'semi-commutative' relation we deduce that all the following traces (by making t_j repeatedly commute backward):
\[
\begin{align*}
\rho_j' &= t_1 t_2 \cdots t_{i-1} t_i t_{i+1} \cdots t_j \cdots t_{j-1} t_{j+1} \cdots t_n \\
\rho_j'_{j-1} &= t_1 t_2 \cdots t_{i-1} t_i t_{i+1} \cdots t_{j-1} t_{j+1} \cdots t_n \\
\rho_j'_{j-2} &= t_1 t_2 \cdots t_{i-1} t_i t_{i+1} \cdots t_{j-2} t_{j+1} \cdots t_n \\
&\vdots \\
\rho_j'_{1} &= t_1 t_2 \cdots t_{i-1} t_i t_{i+1} \cdots t_1 t_{2+1} \cdots t_n
\end{align*}
\]
must violate the property \psi too. However, first(\rho_j') > first(\rho_{\max}). This contradicts the definition of \rho_{\max}.

\[\square\]

V. TRACE INTERPOLATION

In preparation for POR and SI to work together, we now further modify the concept of persistent set so that it applies for any state sharing the same program point (to be compatible with Def 6). The previous definition applies for a specific state only.

**Definition 9 (Semi-Commutative After A Program Point)**. For all \langle t_1, t_2 \rangle \in T and all properties \psi, we say \langle t_1 \rangle semi-commutes with \langle t_2 \rangle after program point pp wrt \ci \psi under the condition \pi, denoted as \langle pp, \pi, t_1 \uparrow t_2, \psi \rangle iff for all feasible derivations from \langle s_0 \rangle \Rightarrow s such that s is at program point pp, if s \models \pi then \langle t_1 \rangle can semi-commute with \langle t_2 \rangle after state s wrt. \ci \psi.

**Definition 10 (Persistent Set Of A Program Point)**. A set \langle T \subseteq \mathcal{T} \rangle of transitions enabled at program point pp is persistent at pp under the trace-interpolant \pi wrt. a property \psi iff, for all feasible derivations \langle s_0 \rangle \Rightarrow s such that s is at program point pp and if s \models \pi then for all feasible derivations s \downarrow t_1 t_2 \cdots t_n \models \pi s_{n+1} s_n including only transitions t_i \in \mathcal{T} and t_i \notin T, 1 \leq i \leq n, each transition in T can semi-commute with t_i after state s wrt. \ci \psi.

Assume that \langle T = \{t_1, t_2, \cdots t_m\} \rangle. The trace interpolant \pi can now be computed as \pi = \bigwedge_i \pi_j \forall 1 \leq j \leq m, 1 \leq i \leq n such that \langle pp, \pi_j, (t_j \uparrow t_i, \psi) \rangle.

One can observe that for each program point, if we attempt to find a smaller persistent set we will indeed strengthen the trace interpolant \pi. In other words, for each program point, it is possible to have different persistent sets associated with different trace interpolants accordingly. In general, a state which satisfies a stronger interpolant will have a smaller persistent set, and therefore, it enjoys more pruning.

We emphasize here the obvious challenge of implementing trace interpolation. This entails obtaining a good estimate of the semi-commutative relation. In general, this task is non-trivial. However, for specific applications, based on our definitions of trace interpolation, it is possible to build customized algorithms whereby information about (property driven) persistent set can be embedded. We elaborate later in section VII.

We also importantly note here that, though a good implementation for computing trace interpolation is not yet generally available, we can readily use any traditional POR method for this approximation. This includes the trivial method in which we estimate the 'semi-commutative' relation as the traditional independence relation (then the trace interpolant is just true). Our formalization of trace interpolation makes this possible as the proposed concepts are strictly weaker than the corresponding concepts used in traditional POR methods. We stress that even in this setting, our combination of state interpolation and POR is in fact better than a simple combination of the two underlying methods (e.g. running each in parallel and seeking the first termination). Again, we elaborate on this later in section VII.

VI. SYNERGY OF STATE AND TRACE INTERPOLATION

We now combine state interpolation with trace interpolation. Here trace interpolants and persistent sets are expected to be computed statically, i.e. by separate analyses. The algorithm is in Figure 6. The function Explore has input s_0 and target \psi. It naturally performs a depth first search of the state space.

**Two Base Cases**: The function Explore handles two base cases. One is when the current state is subsumed by some computed (and memoed) interpolant \pi. No further exploration is needed, and \pi 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**: Transitions which can be scheduled but are disabled at the current state will strengthen the interpolant as in line 6. We then make use of the (static) persistent set T computed for the current program point. The trace interpolant \pi_{trace} associated with T will contribute to the interpolant of the current state too (line 15). In general, customized algorithm to compute \langle T, \pi_{trace} \rangle is needed. We will further comment on this in the next section. Importantly, however, it is possible for us to replace this component by any traditional POR method. T will be the persistent set that method produces while \pi_{trace} will simply be true. Finally, we recursively follow those transitions which belong to T and are enabled at the current state. Note that the persistent set is defined wrt. the program point pp, as long as the current state s \models \pi_{trace}, we can safely ignore those transitions which do not belong to T. However, the context of current state might be too strong, and some transitions in T are no longer enabled in this current state. This is taken care by line 10. The interpolant of each child state contributes to the interpolant of the current state as in line 13. In our current framework, interpolants are propagated back using the precondition operations \pre, where \pre(t, \phi) denote the precondition wrt. the transition t and the postcondition \phi [3]. As discussed earlier, in general, our interpolants are more powerful and can give rise to more pruning than interpolants generated by SMT sover.
Fig. 6 Synergy Algorithm (DFS)

(1) Initially : Explore($s_0$)

function Explore($s$)
(2) if memoed($s, \pi$) return $\pi$
(3) if $s \notin \psi$ Report Error
(4) $\pi = \psi$
(5) foreach $t$ in (Schedulable($s$) \ Enabled($s$)) do
(6) $\pi = \pi \land \pre(t, false)$
endfor
(7) $pp = pc(s)$
(8) $(T, \pi_{trace}) = \text{Persistent Set}(s, pp)$
(9) foreach $t$ in $(T \cap \text{Enabled}(s))$ do
(10) $\pi' = \text{Explore}(s')$
(11) $\pi = \pi \land \pre(t, \pi'')$
endfor
(12) memo and return ($\pi \land \pi_{trace}$)
end function

Fig. 7 Inductive Correctness

Theorem 1. The algorithm in Figure 6 is sound.

Proof Outline. We use structural induction. Refer to Figure 7. Assume that from $s_0$ we reach state $s_1$ such that $pc(s_1) = pp$. W.l.o.g. assume that at $s_i$ there are three transitions which are schedulable, namely $t_1, t_2, t_3$, of which only $t_1$ and $t_2$ are enabled. Also assume that under the trace interpolant $\pi_2$, the persistent set of $pp$, and therefore of $s_i$, is just $\{t_1\}$. From the algorithm, we will extend $s_i$ with $t_1$ (line 11) and attempt to verify the subtree $A$ (line 12). Our induction hypothesis is that we have finished considering $A$, and indeed, it is safe under the interpolant $\pi_A$. That subtree will contribute $\pi_1 = \pre(t_1, \pi_A)$ (line 13) to the interpolant of $s_i$.

Due to the trace interpolant $\pi_2$, the branch having transition $t_2$ followed by the subtree $B$ is pruned and that is safe, due to Lemma 1. Also, the disabled transition $t_3$ contributes $\pi_3$ (line 6) to the interpolant of $s_i$. Now we need to prove that $\pi = \pi_1 \land \pi_2 \land \pi_3$ is indeed a sound interpolant for program point $pp$.

Assume that subsequently in the search, we reach some state $s_j$ such that $pc(s_j) = pp$. We will prove that $\pi$ is a sound interpolant of $pp$ by proving that if $s_j \models \pi$, then the pruning of $s_j$ is safe.

First, $s_j \models \pi$ implies that $s_j \models \pi_3$. Therefore, at $s_j$, $t_3$ is also disabled. Second, assume that $t_1$ is enabled at $s_j$ and $s_j \rightarrow s_{j+1}$ (if not the pruning of $t_1$ followed by $A'$ is definitely safe). Similarly, $s_j \models \pi$ implies that $s_j \models \pi_1$. Consequently, $s_{j+1} \models \pi_A$ and therefore the subtree $A'$ is safe too. Lastly, $s_j \models \pi$ implies that $s_j \models \pi_2$. Thus the reasons which ensure that the traces ending with subtree $A$ cover the traces ending with subtree $B$ also hold at $s_j$. That is, the traces ending with subtree $A'$ also cover the traces ending with subtree $B'$. $\square$

VII. Toward Practical Usage

We proceed in two parts. First, in section VII-A, we consider the implementation of trace interpolation. Though the underlying concept is general and powerful, a general implementation is hard to construct. However, for customized settings, it is possible to create customized algorithms. This is akin to the general concept of dynamic programing or branch-and-bound in algorithms, where there is not one fixed algorithm, but rather, the concept guides the construction of a custom algorithm. We present three examples.

Second, in section VII-B, we demonstrate the synergy of state-based and trace-based reduction. For simplicity, we decide to experiment on models instead of real concurrent programs written in C/C++. To close this gap, however, we can always employ the same techniques employed by Inspect [21]. The key message here is that, even when state-based and trace-based methods are not effective individually, our combined framework still offers significant reduction.

A. Customized Trace Interpolation

Resource Usage of Concurrent Programs: Programs make use of limited resource (such as time, memory, bandwidth). Validation of resource usage of sequential setting is already a hard problem. It is even more challenging in the setting of concurrent programs, where the search space, due to interleaving, is astronomically huge.

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 a guard condition of any transition. The property to be verified is that, “at all times, $R$ is (up-) bounded by some constant”.

Proposition 1. Given a concurrent program, resource variable $R$, and safety property $\psi \equiv R \leq C$, where $C$ is a constant. For all un-guarded transitions (the guard conditions are true) $t_1 : R = R + c_1, t_2 : R = R + 2, t_3 : R = R - c_2$ where $c_1, c_2 > 0$, we have for all program point $pp$:

- $\langle pp, true, t_1 \uparrow t_2, \psi \rangle$
- $\langle pp, true, t_1 \uparrow t_3, \psi \rangle$
- $\langle pp, true, t_2 \uparrow t_3, \psi \rangle$

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.

The consequence is clear. For this class of problem, we only need to exploit the interleaving space caused by the existence of transition which is guarded and manipulates the resource.

As a result, we cannot model the behavior of a typical garbage collector.
variable at the same time. Without such a transition, only one complete trace is required to be explored.

**Fig. 8** Two closely coupled processes (a) and performance of our Trace Interpolation

![Image of two closely coupled processes](image)

**Example 4.** Let us refer back to the example of two closely coupled processes introduced in section II. Figure 8 shows again the control flow graphs of two processes, but now under the assumption that $x$ is the resource variable of interest.

The safety property and relevant transitions are depicted in the figure. Although this example is very simple, it illustrates one of the most common (and general) type of data races in concurrent programs.

We can clearly see that the program is safe wrt. specified safety property $\psi$. From Proposition 1, we require to explore only one complete trace to prove this safety.

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 terminating states). SI alone can reduce the search space in this example, and requires to explore only 9 states and 4 subsumed states (as in section II). Symbolic methods as in [6], [20] also will not perform well with this example if we also consider the search space generated by solver. That is because a general solver has no mechanism to capture the fact that some transitions are relevant to the proof of the property at hand, though their orders are not. For a similar reason, without explicitly dealing with the concept of swapping, [8] will not achieve the level of efficiency we achieve with our method.

**Detection of Race Conditions:** [21] proposed a property driven pruning algorithm to detect race conditions in multi-threaded programs. This work has achieved more reduction in comparison with DPOR. The key observation is that, at a certain location (program point) $pp$, 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 [21], 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, [21] is indeed a (conservative) realization of our trace interpolation, specifically targeted for detection of race conditions. Their mechanism to capture the trace interpolants 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 cannot be unset. It is then easy to see that for all program point $pp$ and transitions $t_1$, $t_2$, we have $\langle pp, flag = 1, t_1 \uparrow t_2, \psi \rangle$. Though simple, this observation will bring us more reduction compared to traditional POR methods.

**B. Synergy of State and Trace Interpolation**

Since a general implementation to compute trace interpolation (TI) is not yet available, we indeed use the most recent implementation of Partial Order Reduction [1] for this component (this is only made possible by our formulation of TI). 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). All prototypes presented here are implemented using the CLP(R) system [10]. 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 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 always less than $N + 2^N$.

Table I is about this example. The performances presented in Table I clearly demonstrate 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 customized trace interpolation requires only a single trace in order to prove safety.

<table>
<thead>
<tr>
<th>$N$</th>
<th>States</th>
<th>$\text{POR}$</th>
<th>States</th>
<th>$\text{SI}$</th>
<th>States</th>
<th>$\text{POR+SI}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>445</td>
<td>0.97</td>
<td>514</td>
<td>0.17</td>
<td>85</td>
<td>0.03</td>
</tr>
<tr>
<td>3</td>
<td>18745</td>
<td>2.73</td>
<td>6562</td>
<td>2.43</td>
<td>455</td>
<td>0.19</td>
</tr>
<tr>
<td>4</td>
<td>896418</td>
<td>886.09</td>
<td>76546</td>
<td>37.53</td>
<td>2333</td>
<td>1.07</td>
</tr>
<tr>
<td>5</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>11275</td>
<td>5.76</td>
</tr>
<tr>
<td>6</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>53261</td>
<td>34.50</td>
</tr>
<tr>
<td>7</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>25775</td>
<td>115.42</td>
</tr>
</tbody>
</table>

**TABLE I**

**The Producer/Consumer Example**

To further demonstrate the power of our synergy framework as well as the power of our trace interpolation concept, we experiment next on the sum-of-ids program. Here, each process
The table shows that POR does not result in any pruning for this benchmark. However, SI, and therefore POR+SI significantly prune the search space. Here we comment that the SI contribution is due to our specific interpolation algorithm; using a generic SMT solver would not achieve the same reduction. Finally, this example can also be translated to resource usage problem, our use of property-driven POR again requires one single trace to prove safety.

\[ \text{POR+SI} = \text{SI} \]

We presented a framework for exploring and pruning the space of interleavings of a concurrent system, pursuant to a target property. In our framework, we first introduced a new notion of trace interpolation in order to capture the reasoning capability of POR, but importantly, this time potentially in regard to the target property. We briefly showed how our theory on trace interpolation can be customized for specific applications. But the main contribution is that the framework combines the use of trace-based reduction techniques with the more established notion of state interpolant used in general-purpose SMT solvers as well as CEGAR. This combination is synergistic in the sense that we obtain more pruning in the combination than if we had applied both state and trace interpolation in separate phases.

\[ T(s) \]

\[ \text{POR} \]

\[ \text{POR+SI} \]

\[ T(s) \]

\[ \text{SI} \]

<table>
<thead>
<tr>
<th>POR = None</th>
<th>POR+SI = SI</th>
</tr>
</thead>
<tbody>
<tr>
<td>( N ) States</td>
<td>( t(s) )</td>
</tr>
<tr>
<td>2</td>
<td>2676</td>
</tr>
<tr>
<td>4</td>
<td>149920</td>
</tr>
<tr>
<td>12</td>
<td>–</td>
</tr>
<tr>
<td>14</td>
<td>–</td>
</tr>
<tr>
<td>14</td>
<td>–</td>
</tr>
</tbody>
</table>

\[ \text{TABLE II} \]

\[ \text{SUM-OF-IDS EXAMPLE} \]

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 [13] has used this to demonstrate some benefits from combining POR and SI.

The first safety property used in [13], namely ‘it is not the case 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 III demonstrates the significant improvements of POR+SI over POR alone and SI alone. We note that the performance of our POR+SI algorithm is better than in [13].

\[ \text{TABLE III} \]

\[ \text{THE DINING PHILOSOPHERS EXAMPLE: PROPERTY (A)} \]

We additionally considered a second safety property as in [13], 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 [13] requires even more time compared to proving property (a). This illustrates the scalability issue of [13], which is representative for other techniques employing general-purpose SMT solver for symbolic pruning (as we carefully discussed in section I).

Finally, to further highlight the power of symbolic execution and the synergy of POR and SI in our framework, we perform experiments on the “Bakery” algorithm. We note that, due to existence of infinite domain variables, model checking cannot handle Bakery algorithm. The results are shown in Table IV.