# Peephole Partial Order Reduction

Chao Wang[1], Zijiang Yang[2], Vineet Kahlon[1], and Aarti Gupta[1]

[1] NEC Laboratories America, Princeton, NJ
{chaowang,kahlon,agupta}@nec-labs.com
[2] Western Michigan University, Kalamazoo, MI
zijiang.yang@wmich.edu

**Abstract.** We present a symbolic dynamic partial order reduction (POR) method for model checking concurrent software. We introduce the notion of guarded independent transitions, i.e., transitions that can be considered as independent in certain (but not necessarily all) execution paths. These can be exploited by using a new peephole reduction method. A symbolic formulation of the proposed peephole reduction adds concise constraints to allow automatic pruning of redundant interleavings in an SMT/SAT solver based search. Our new method does not directly correspond to any explicit-state algorithm in the literature, e.g., those based on persistent sets. For two threads, our symbolic method guarantees the removal of all redundant interleavings (better than the smallest persistent-set based methods). To our knowledge, this type of reduction has not been achieved by other symbolic methods.

## 1 Introduction

Verifying concurrent programs is hard due to the large number of interleavings of transitions from different threads. In explicit-state model checking, partial order reduction (POR) techniques [7, 17, 20] have been be used to exploit the equivalence of interleavings of independent transitions to reduce the search state space. Since computing the precise dependence relation may be as hard as verification itself, existing POR methods often use a conservative static analysis to compute an approximation. Dynamic partial order reduction [6] and Cartesian partial order reduction [11] lift the need of applying static analysis *a priori* by detecting collision (data dependency) on-the-fly. These methods in general can achieve more reduction due to the more accurate collision detection. However, applying these POR methods (which were designed for explicit-state algorithms) to symbolic model checking is not an easy task.

A major strength of SAT-based symbolic methods [2] is that *property dependent* and *data dependent* search space reduction is automatically exploited inside modern SAT or SMT (Satisfiability Modulo Theory) solvers, through the addition of conflict clauses and non-chronological backtracking. Symbolic methods are often more efficient in reasoning about variables with large domains. However, combining classic POR methods (e.g., those based on persistent-sets [8]) with symbolic algorithms has proven to be difficult [1, 15, 10, 3, 13]. The difficulty arises from the fact that symbolic methods typically manipulate a large *set of states* implicitly as opposed to manipulating states individually. Capturing and exploiting transitions that are dynamically independent with respect to a *set of states* is much harder than it is for individual states.

C.R. Ramakrishnan and J. Rehof (Eds.): TACAS 2008, LNCS 4963, pp. 382–396, 2008.

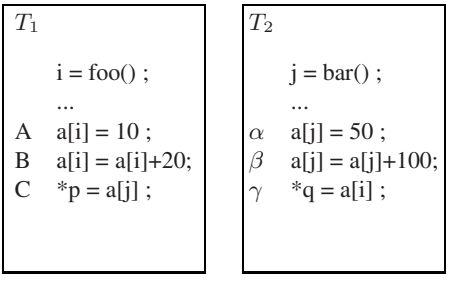



**Fig. 1.** $t_A, t_B$ are independent with $t_\alpha, t_\beta$ when $i \neq j$; $t_C$ is independent with $t_\gamma$ when $(p \neq q)$.

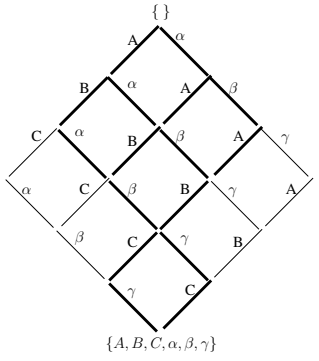



**Fig. 2.** The lattice of interleavings

For example, in Fig. 1 there are two concurrent threads accessing a global array $a[\,]$. The two pointers $p$ and $q$ may be aliased. Statically, transitions $t_A, t_B$ in thread $T_1$ are dependent with $t_\alpha, t_\beta$ in $T_2$. Therefore, POR methods relying on a static analysis may be ineffective. Note that when $i \neq j$ holds in some executions, $t_A, t_B$ and $t_\alpha, t_\beta$ become independent, meaning that the two sequences $t_A; t_B; t_\alpha; t_\beta; t_C; t_\gamma$; and $t_\alpha; t_\beta; t_A; t_B; t_C; t_\gamma$; are equivalent. However, none of the existing symbolic partial order reduction methods [1, 15, 10, 3, 13] takes advantage of such information[1]. Among explicit-state POR methods, dynamic partial order reduction [6] and Cartesian partial order reduction [11] are able to achieve some reduction by detecting conflicts on-the-fly; in any individual state $s$, the values of $i$ and $j$ (as well as $p$ and $q$) are fully determined, making it much easier to detect conflicts. However, it is not clear how to directly apply these techniques to symbolic model checking, where conflict detection is performed with respect to a set of states.

Missing out these kind of partial-order reductions can be costly, since the symbolic model checker needs to exhaustively search among the reduced set of execution sequences. The number of valid interleavings (sequences) can be large even for moderate sized programs. For the running example, we can capture all possible interleavings using a lattice structure (Fig. 2). Let $Q = \{t_A, t_B, t_C, t_\alpha, t_\beta, t_\gamma\}$ be the set of transitions from both threads. Each vertex in the figure represents a distinct subset of $Q$, consisting of the executed transitions up to that point. The top vertex is $\{\ \}$ and the bottom vertex is $\{t_A, t_B, t_C, t_\alpha, t_\beta, t_\gamma\}$. A path from top to bottom denotes a unique interleaving. For example, the left-most path corresponds to $t_A; t_B; t_C; t_\alpha; t_\beta; t_\gamma$. The set of vertices forms a powerset $2^Q$.

In this paper, we present a new *peephole partial order reduction* method to exploit the dynamic independence of transitions. To this end, we introduce a new notion of independence relation called *guarded independence relation (GIR)*. It is an extension of

[1] The method in [13] can reduce equivalent interleavings if we replace `i=foo()` and `j=bar()` with `i=1` and `j=2`, but not in the general case.

the classic (conditional) independence relation [14, 8]: instead of defining independence with respect to either a single state or for all global states, we define the GIR relation $R_G$ with respect to a predicate over programs variables. Each $\langle t_1, t_2, c_G\rangle \in R_G$ corresponds to a pair of transitions $t_1, t_2$ that are independent iff $c_G$ holds. A major advantage of GIR is that it can be accurately computed by a simple traversal of the program structure. We further propose a peephole reduction which concisely captures the guarded independent transitions as constraints over a fixed number of adjacent transitions to restrict the satisfiability formula during symbolic search (e.g., in bounded model checking). The added constraints allow the underlying SAT/SMT solver to prune search space automatically. Adding these GIR constraints requires identification of a pattern in a fixed sized time window only.

The basic observation exploited by various POR methods is that different execution sequences may correspond to the same equivalence class. According to Mazurkiewicz's trace theory [16], two sequences are equivalent if they can be obtained from each other by successively permuting adjacent independent transitions. In this sense, our peephole POR method has the same goal as the classic POR methods[7, 17, 20, 6, 11]; however, it does not directly correspond to any existing method. In particular, it is not a symbolic implementation of any of these explicit-state methods. For a system with two threads, our method can guarantee optimality in reduction; that is, all redundant interleavings are removed (proof is in Section 3.2). To our knowledge, there has not been such guarantee among existing POR methods. We also show an example on which our method achieves strictly more reduction than any persistent-set based method. Finally, the proposed encoding scheme is well suited for symbolic search using SAT/SMT solvers.

To summarize, our main contributions are: (1) the notion of guarded independence relation, which accurately captures independence between a pair of transitions in terms of predicates on states; (2) a peephole partial order reduction that adds local constraints based on the guarded independence relation, along with a symbolic formulation; (3) the guarantee of removing all redundant interleavings for systems with two threads. This kind of reduction has not been achieved by previous symbolic methods [1, 15, 10, 3, 13].

## 2 Guarded Independence Relation

In this section, we review the classic notion [14, 8] of independent transitions, and then present the new notion of guarded independence relation.

Let $T_i$ $(1 \leq i \leq N)$ be a thread with the set $trans_i$ of transitions. Let $trans = \bigcup_{i=1}^{N} trans_i$ be the set of all transitions. Let $V_i$ be the set of local variables in thread $T_i$, and $V_{global}$ be the set of global variables. For $t_1 \in trans_i$, we denote the thread index by $tid_{t_1}$, and denote the enabling condition by $en_{t_1}$. If $t_1$ is a transition in $T_i$ from control location $loc_1$ to $loc_2$ and is guarded by $cond$, then $en_{t_1}$ is defined as $(pc_i = loc_1) \wedge cond$. Here $pc_i \in V_i$ is a special variable representing the thread program counter. Let $S$ be the set of global states of the system. A state $s \in S$ is a valuation of all local and global variables. For two states $s, s' \in S$, $s \xrightarrow{t_1} s'$ denotes a state transition by applying $t_1$, and $s \overset{t_i \ldots t_j}{\Rightarrow} s'$ denotes a sequence of state transitions.

### 2.1 Independence Relation

**Definition 1 (Independence Relation [14, 8]).** *$R \subseteq trans \times trans$ is an independence relation iff for each $\langle t_1, t_2 \rangle \in R$ the following two 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 $t_1, t_2$ are enabled in $s$, there is a unique state $s'$ such that $s \overset{t_1 t_2}{\Rightarrow} s'$ and $s \overset{t_2 t_1}{\Rightarrow} s'$.*

In other words, independent transitions can neither disable nor enable each other, and enabled independent transitions commute. As pointed out in [7], the definition has been mainly of semantic use since it is not practical to check the above two properties for all states to determine which transitions are independent. Instead, traditionally collision detection often uses conservative but easy-to-check sufficient conditions. For instance, the following properties [7] have been used in practice to compute independent transitions:

1. the set of threads that are active for $t_1$ is disjoint from the set of threads that are active for $t_2$, and
2. the set of objects that are accessed by $t_1$ is disjoint from the set of objects that are accessed by $t_2$.

Note that some independent transitions may be conservatively classified as dependent, like $t_1{:}a[i] = e_1$ and $t_2{:}a[j] = e_2$ when $i \neq j$, since it is not clear statically if $a[i]$ and $a[j]$ refer to the same element. This can in turn lead to a coarser persistent set.

In the *conditional* dependence relation [14, 8], two transitions are defined as independent with respect to a state $s \in S$ (as opposed to for all $s \in S$). This extension is geared towards explicit-state model checking, in which persistent sets are computed for individual states. A persistent set at state $s$ is a subset of the enabled transitions that need to be traversed in adaptive search. A transition is added to the persistent set if it has any conflict with a future operation of another thread. The main difficulty in persistent set computation lies in detecting future collision with enough precision. Although it is not practical to compute the conditional dependence relation for each state in $S$ for collision detection purposes, there are explicit-state methods (e.g., [6, 11]) to exploit such dynamically independent transitions. However, these classic definitions of independence are not well suited for symbolic search.

### 2.2 Guarded Independence Relation

**Definition 2.** *Two transitions $t_1, t_2$ are guarded independent with respect to a condition $c_G$ iff $c_G$ implies that the following properties hold:*

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 $t_1, t_2$ are enabled in $s$, there is a unique state $s'$ such that $s \overset{t_1 t_2}{\Rightarrow} s'$ and $s \overset{t_2 t_1}{\Rightarrow} s'$.*

This can be considered as an extension of the *conditional* dependence relation; instead of defining $\langle t_1, t_2, s \rangle$ with respect to a state $s \in S$, we define $\langle t_1, t_2, c_G \rangle$ with respect to a predicate over local and global program variables. The independence relation is valid for all states in which $c_G$ holds, i.e., it is valid with respect to a (potentially large) set of

states. Unlike the previous definitions, when computing GIR, we are able to apply the two properties in Definition 2 precisely.

The guard $c_G$ can be efficiently computed by a traversal of the structure of the program. For a transition $t$, we use $V_{RD}(t)$ to denote the set of variables read by $t$, and $V_{WR}(t)$ to denote the set of variables written by $t$. We define the *potential conflict set* between $t_1$ and $t_2$ from different threads to be

$$\mathcal{C}_{t_1,t_2} = V_{RD}(t_1) \cap V_{WR}(t_2) \cup V_{RD}(t_2) \cap V_{WR}(t_1) \cup V_{WR}(t_1) \cap V_{WR}(t_2) \ .$$

In our running example, $\mathcal{C}_{t_A,t_\alpha} = \{a[i], a[j]\}$. For a C-like program, we list the different scenarios under which we compute the guarded independence relation $R_G$:

1. when $\mathcal{C}_{t_1,t_2} = \emptyset$, add $\langle t_1, t_2, true\rangle$ to $R_G$;
2. when $\mathcal{C}_{t_1,t_2} = \{a[i], a[j]\}$, add $\langle t_1, t_2, i \neq j\rangle$ to $R_G$;
3. when $\mathcal{C}_{t_1,t_2} = \{*p_i, *p_j\}$, add $\langle t_1, t_2, p_i \neq p_j\rangle$ to $R_G$;
4. when $\mathcal{C}_{t_1,t_2} = \{x\}$, consider the following cases:
   a. **RD-WR:** if $x \in V_{RD}(t_1)$ and the assignment $x := e$ appears in $t_2$, add $\langle t_1, t_2, x = e\rangle$ to $R_G$;
   b. **WR-WR:** if $x := e_1$ appears in $t_1$ and $x := e_2$ appears in $t_2$, add $\langle t_1, t_2, e_1 = e_2\rangle$ to $R_G$;
   c. **WR-C:** if $x$ appears in the condition $cond$ of a branching statement $t_1$, such as `if(cond)`, and $x := e$ appears in $t_2$, add $\langle t_1, t_2, cond = cond[x \rightarrow e]\rangle$ to $R_G$, in which $cond[x \rightarrow e]$ denotes the replacement of $x$ with $e$.

Overall the computational complexity is $O(|trans|^2)$, where $|trans|$ is the number of transitions. If desired, the set of rules can be easily extended to handle a richer set of language constructs.

Rules 1,2, and 3 correspond to standard semantics of a program. Pattern 4(a) states that two read/write operations to the same variable are guarded independent if the write operation does not change its value. Pattern 4(b) states that two write operations to the same variable are guarded independent if their newly assigned values are the same. In these two cases, $c_G$ may evaluate to true more frequently than one may think, especially when these variables have small ranges and when they are used for branching purposes. If $b$ is a Boolean variable, then $b := e_1$ and $b := e_2$ independent in two of the four possible cases. Pattern 4(c) is a special case of 4(a): clearly $x = e$ implies $cond = cond[x \rightarrow e]$; however, there are cases when $x \neq e$ but $cond = cond[x \rightarrow e]$. For example, let `if(`$x < 10$`)` be a transition in thread 1 and $x := e$ be in thread 2. They are guarded independent if $(x < 10) = (e < 10)$, even if $x$ changes after the assignment. Multiple patterns can appear in the same pair of transitions. In such cases, $c_G$ is a conjunction or disjunction of individual conditions. For example, consider $t_1$:`if(a[i]>5)` and $t_2$:`a[j]:=x`. Here $c_G$ is defined as $i \neq j \vee ((a[i] > 5) = (x > 5))$.

In symbolic search based on SMT/SAT solvers, the guarded independence relation can be compactly encoded as symbolic constraints in the problem formulation, as described in the next section. These constraints facilitate automatic pruning of the search space.

## 3 Peephole Partial Order Reduction

After reviewing the basics of SMT/SAT based bounded model checking in Section 3.1, we will present our new partial order reduction method in Section 3.2.

### 3.1 Bounded Model Checking (BMC)

Given a multi-threaded program and a reachability property, BMC can check the property on all execution paths of the program up to a fixed depth $K$. For each step $0 \leq k \leq K$, BMC builds a formula $\Psi$ such that *$\Psi$ is satisfiable iff there exists a length-$k$ execution that violates the property*. The formula is denoted $\Psi = \Phi \wedge \Phi_{prop}$, where $\Phi$ represents all possible executions of the program up to $k$ steps and $\Phi_{prop}$ is the constraint indicating violation of the property. (For more information about $\Phi_{prop}$, refer to [2].) In the following, we focus on the formulation of $\Phi$.

Let $V = V_{global} \cup \bigcup V_i$, where $V_{global}$ are global variables and $V_i$ are local variables in $T_i$. For every local (global) program variable, we add a state variable to $V_i$ ($V_{global}$). Array and pointer accesses need special handling. For an array access $a[i]$, we add separate variables for the index $i$ and for the content $a[i]$. Similarly, for a pointer access $*p$, we assign separate state variables for $(*p)$ and $p$. We add a $pc_i$ variable for each thread $T_i$ to represent its current program counter. To model nondeterminism in the scheduler, we add a variable $sel$ whose domain is the set of thread indices $\{1, 2, \ldots, N\}$. A transition in $T_i$ is executed only when $sel = i$.

At every time frame we add fresh copies of the set of state variables. Let $v^i \in V^i$ denote the copy of $v \in V$ at the $i$-th time frame. To represent all possible length-$k$ interleavings, we first encode the transition relations of individual threads and the scheduler, and unfold the composed system exactly $k$ time frames.

$$\Phi := \; I(V^0) \wedge \bigwedge_{i=0}^{k} \left( SCH(V^i) \wedge \bigwedge_{j=1}^{N} TR_j(V^i, V^{i+1}) \right)$$

where $I(V^0)$ represents the set of initial states, $SCH$ represents the constraint on the scheduler, and $TR_j$ represents the transition of thread $T_j$. Without any partial order reduction, $SCH(V^i) := true$, which means that $sel$ takes arbitrary values at every step. This default $SCH$ considers all possible interleavings. Partial order reduction can be implemented by adding constraints to $SCH$ to remove redundant interleavings.

We now consider the formulation of $TR_j$. Let $VS_j = V_{global} \cup V_j$ denote the set of variables visible to $T_j$. At the $i$-th time frame, for each $t \in trans_j$ (a transition between control locations $loc_1$ and $loc_2$), we create $tr_t^i$. If $t$ is an assignment $v := e$, then $tr_t^i :=$

$$pc_j^i = loc_1 \wedge pc_j^{i+1} = loc_2 \wedge v^{i+1} = e^i \wedge (VS_j^{i+1} \setminus v^{i+1}) = (VS_j^i \setminus v^i) \;\; .$$

If $t$ is a branching statement[2] $assume(c)$, as in `if(c)`, then $tr_t^i :=$

$$pc_j^i = loc_1 \wedge pc_j^{i+1} = loc_2 \wedge c^i \wedge VS_j^{i+1} = VS_j^i.$$

[2] We assume that there is a preprocessing phase in which the program is simplified to have only assignments and branching statements, as in tools like FSoft [12].

Overall, $TR_j^i$ is defined as follows:

$$TR_j^i := \left( sel^i = j \wedge \bigvee_{t \in trans_j} tr_t^i \right) \vee \left( sel^i \neq j \wedge V_j^{i+1} = V_j^i \right)$$

The second term says that if $T_j$ is not selected, variables in $V_j$ do not change values.

### 3.2 Peephole Partial Order Reduction

$SCH$ initially consists of all possible interleavings of threads. If multiple length-$k$ sequences are in the same equivalence class, only one representative needs to be checked for property violation. To facilitate such reduction, we add constraints for each pair of guarded independent transitions to restrict the scheduler.

For each $\langle t_1, t_2, c_G \rangle \in R_G$ such that $tid_{t_1} < tid_{t_2}$, we conjoin the following constraint to $SCH$,

$$en_{t_1}(V^k) \wedge en_{t_2}(V^k) \wedge c_G(V^k) \rightarrow \neg(sel^k = tid_{t_2} \wedge sel^{k+1} = tid_{t_1})$$

Here, $en_{t_1}(V^k)$ and $en_{t_2}(V^k)$ are the enabling conditions for $t_1$ and $t_2$ at the $k$-th time frame. The above constraint says that, if independent transitions $t_1$ and $t_2$ are enabled, sequences starting with both $t_1; \ldots$ and $t_2; \ldots$ are allowed to be explored. However, among the sequences starting with $t_2; \ldots$, we forbid $\underline{t_2; t_1;}$... through the addition of constraint $\neg(sel^k = tid_{t_2} \wedge sel^{k+1} = tid_{t_1})$. In essence, the above constraint enforces a fixed order on the priority of scheduling two independent transitions $t_1, t_2$. We always prefer sequences in which two *adjacent* independent transitions $t_1, t_2$ are scheduled in their thread index order, i.e., $t_1$ ahead of $t_2$ if $tid_{t_1} < tid_{t_2}$. The alternative sequences are removed, as illustrated in Fig. 3.

In the running example, the new $SCH(V^k)$ is

$$\begin{array}{l}
\big(pc_1^k = A \wedge pc_2^k = \alpha \wedge (i^k \neq j^k) \rightarrow \neg(sel^k = 2 \wedge sel^{k+1} = 1)\big) \wedge \\
\big(pc_1^k = A \wedge pc_2^k = \beta \wedge (i^k \neq j^k) \rightarrow \neg(sel^k = 2 \wedge sel^{k+1} = 1)\big) \wedge \\
\big(pc_1^k = B \wedge pc_2^k = \alpha \wedge (i^k \neq j^k) \rightarrow \neg(sel^k = 2 \wedge sel^{k+1} = 1)\big) \wedge \\
\big(pc_1^k = B \wedge pc_2^k = \beta \wedge (i^k \neq j^k) \rightarrow \neg(sel^k = 2 \wedge sel^{k+1} = 1)\big) \wedge \\
\big(pc_1^k = C \wedge pc_2^k = \gamma \wedge (i^k \neq j^k \wedge p^k \neq q^k) \rightarrow \neg(sel^k = 2 \wedge sel^{k+1} = 1)\big)
\end{array}$$

When $i \neq j$, the above constraint removes sequences containing $\underline{t_\alpha; t_A;} \ldots$.

**Theorem 1.** *All interleavings removed by the peephole reduction are redundant.*

*Proof.* Let $\pi = \pi_0 \pi_1 ...$ be a valid sequence that is forbidden by the peephole reduction. Then there exists at least one index $k$ in $\pi$ such that $c_G$ holds, $\pi_k$ and $\pi_{k+1}$ are independent. By swapping the two adjacent independent transitions, we produce another sequence $\pi'$ such that $\pi'_k = \pi_{k+1}$, $\pi'_{k+1} = \pi_k$, and $\pi'_i = \pi_i$ for all $i \neq k$ and $i \neq k+1$. $\pi'$ is (Mazurkiewicz) equivalent to $\pi$. (1) If $\pi'$ is not forbidden by the peephole reduction, $\pi$ is redundant and we have a proof. (2) If $\pi'$ is also forbidden, then there exists an index $k$ in $\pi'$ such that $\pi'_k, \pi'_{k+1}$ are guarded independent—because otherwise $\pi'$ cannot be removed by the peephole reduction. Due to the finiteness of the sequence, if

we continue the *find-and-swap* process, eventually we will find a sequence $\pi''$ that is not forbidden by the peephole reduction. In this case $\pi$ is redundant (equivalent to $\pi''$) and we complete the proof. □

**Theorem 2.** *For two threads, the peephole reduction removes all the redundant interleavings.*

*Proof.* We prove by contradiction. Assume $\pi, \pi'$ are two remaining sequences and they are (Mazurkiewicz) equivalent. By definition, $\pi$ and $\pi'$ have the same set of transitions; $\pi'$ is a permutation of $\pi$. Let $\pi_j$ in $\pi$ be the first transition from $T_2$ that is swapped to be $\pi'_i$ in $\pi'$ (where $i < j$). Then $\pi$ and $\pi'$ share a common prefix up to $i$ (Fig. 4). Furthermore, all transitions $\pi_i, \ldots, \pi_{j-1}$ in $\pi$ belong to $T_1$. This is because if any of them belongs to $T_2$, it would not be possible to move $\pi_j$ ahead of $\pi'$ – the order of transitions from the same thread cannot be changed. Therefore, there are only two cases regarding $\pi_{j-1}$ from $T_1$ and $\pi_j$ from $T_2$: (1) if they are dependent, swapping them produces a non-equivalent sequences; (2) if they are independent, the fact that $\pi_j$ appears after $\pi_{j-1}$ in $\pi$ means that $tid_{\pi_j} > tid_{\pi_{j-1}}$. This implies that $tid_{\pi_j} > tid_{\pi_i}$, and $\pi'$ would have been removed. Since both cases contradict the assumption, the assumption is not correct. □

For more than two threads, the proposed peephole reduction does not always guarantee the removal of all redundant interleavings. For example, let transitions $t_A, t_\alpha, t_x$ belong to threads $T_1, T_2, T_3$, respectively. Assume that $t_A$ and $t_x$ are dependent, but $t_\alpha$ is guarded independent with both $t_A$ and $t_x$. When the guard is true, the following two interleavings are equivalent,

$$\underline{t_x; t_A;} t_\alpha; \ldots$$
$$t_\alpha; \underline{t_x; t_A;} \ldots$$

Both sequences conform to the GIR constraints, since the segment $(\underline{t_A;} t_\alpha;)$ conforms to $tid_{t_A} < tid_{t_\alpha}$ and the segment $(t_\alpha; \underline{t_x;})$ conforms to $tid_{t_\alpha} < tid_{t_x}$. The three transitions can be grouped into two independent sets: $\{t_A, t_x\}$ and $\{t_\alpha\}$. The non-optimality arises from the fact that there is not an order of the two sets in which the pairwise independent

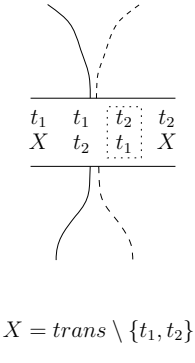



**Fig. 3.** We remove only redundant interleavings

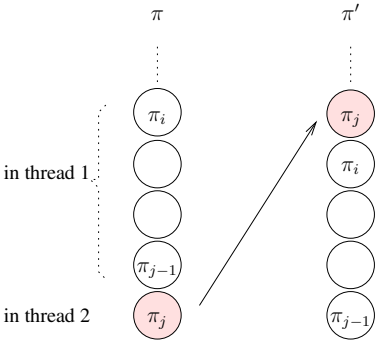



**Fig. 4.** For two threads, we remove all redundant interleavings

transitions are ordered in a way consistent with the ordered thread indices[3] . If we arrange the threads in a different order, $t_A, t_x, t_\alpha$ are in $T_1, T_2, T_3$, then the sequence $t_\alpha; \underline{t_x; t_A;} \ldots$ would be removed by our reduction. Extending the peephole reduction to guarantee the removal of all redundant interleavings in the more general cases may be possible. However, any such extension is likely to be more expensive than the peephole reduction proposed here. In practice, there is a tradeoff between the encoding overhead and the amount of achievable reduction.

### 3.3 Comparison with Persistent-Set Based Methods

The peephole reduction add only local constraints, i.e., constraints over a fixed number of adjacent time steps. The reduction relies on whether two transitions $t_1, t_2$ are *locally* (guarded) independent, for which the precise information is available (Section 2.2). This is in contrast to persistent set based methods relying on detecting *future* conflicts (for which precise information in general is expensive to compute). Persistent set based methods were designed to be used in adaptive search during explicit-state model checking. Depending on the order in which transitions are picked during persistent set computation, there can be more than one persistent set. Some persistent sets achieve more reduction than others. In practice, computing the smallest persistent set at each step of the adaptive search can be costly. The following example shows that even if the smallest persistent sets were available at each step of the adaptive search (in a hypothetical algorithm), there would still be redundant interleavings.

Fig. 5 is derived from the running example by assuming $i = 1$, $j = 2$, and $p \neq q$. Since $t_A$ has a collision with a future transition of thread $T_2$ (transition $t_\gamma$), and similarly $t_\alpha$ has a collision with $t_C$, the smallest persistent set at the starting point is $PS(s) = \{t_A, t_\alpha\}$. This allows both $t_A; \ldots$ and $t_\alpha; \ldots$ to be explored. In the reduced lattice in Fig. 5, there are many redundant sequences (paths over solid thick lines). In fact, the only reduction is achieved by the persistent set $PS(s') = \{C\}$, which is a strict subset of the enabled set $\{C, \gamma\}$.

In our peephole POR method, since $t_A$ and $t_\alpha$ are independent and $tid_{t_A} < tid_{t_\alpha}$, we remove the sequences starting with $\underline{t_\alpha; t_A;}$... but allow the sequences starting with $t_\alpha; t_\beta;$.... As shown by the reduced lattice in Fig. 6, the following adjacent transitions are forbidden: $(t_\alpha, t_A)$, $(t_\beta; t_A)$, $(t_\alpha; t_B)$, $(t_\beta; t_B)$, and $(t_\gamma; t_C)$. The forbidden combinations of adjacent independent transitions are depicted by dotted arrows. The last dotted arrow in Fig. 6 deserves more explanation: our method forbids $t_C$ only when $t_\gamma$ is the previous transition, but allows $t_C$ to execute if the previous transition is $t_B$ (both $t_C$ and $t_B$ are from $T_1$). Note that there is no redundant interleaving.

This example suggests that the benefit of *peephole reduction* is separate from the benefit of accurate *guarded independence relation*. In this example, both the persistent set method and the peephole reduction can use the most precise independence relation, yet our method can forbid more interleavings. Therefore, the advantages of our approach in general come from two distinct sources: the peephole reduction and the accurate guarded independence relation.

[3] There are eight distinct scenarios of the pairwise dependency of $\langle t_A, t_\alpha, t_x \rangle$, each of which corresponds to six interleavings. The proposed reduction removes all the redundant sequences except one.

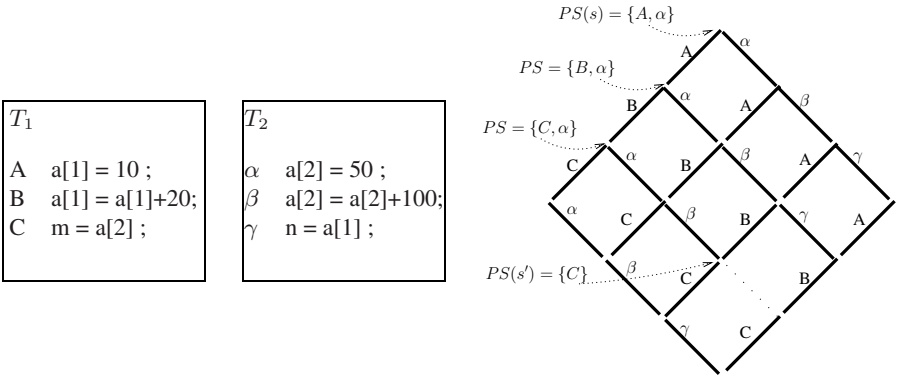
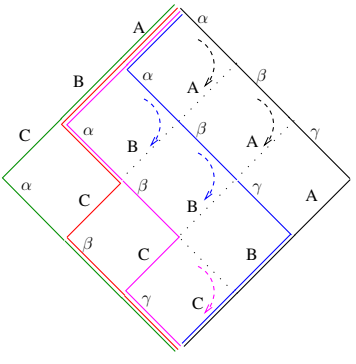



**Fig. 5.** Smallest persistent-sets do not remove all redundant interleavings



Our method add constraints to $SCH$ to disallow sequences containing the following fragments:

$\alpha; A;$
$\beta; A;$
$\alpha; B;$
$\beta; B;$
$\gamma; C;$

**Fig. 6.** Our method removes all the redundant interleavings

Persistent set computation looks only into *current* and *future* transitions of other threads. The methods based on sleep set [9] also consider *past* transitions when computing reduction. Our peephole POR method differs from sleep sets in two aspects: First, the peephole reduction guarantees the optimality for two threads. Second, the encoding in peephole reduction is *memoryless* in that there is no need to store information about past transitions (and to carry the information around) explicitly.

## 4 Reducing the Overhead of GIR Constraints

For the symbolic formulation outlined in the previous section, the number of GIR constraints added is linear in the number of guarded independent transition pairs (which can be quadratic in the number of transitions). These constraints need to be added at each time frame, which may pose a significant overhead for the SMT/SAT solver. On the other hand, missing out these reductions can be costly, since the model checker needs to explore all allowed execution sequences, which can be many. In practice, there is a tradeoff between the encoding overhead and the amount of achievable reduction. Having said that, there are techniques to reduce the encoding overhead in practical settings.

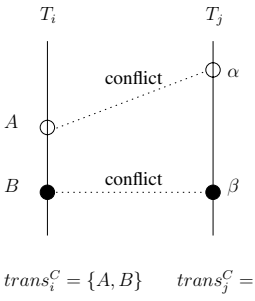



**Fig. 7.** Using dependent transitions to simplify encoding

First, if a cheap static analysis can be used to figure out statically independent transitions, based on which high quality persistent sets can be computed, then it should be used before proceeding to the more advanced reduction. In principle, one can reserve peephole reduction to transitions that are not statically independent (or those that cannot be easily identified statically). Furthermore, when programs have clearly specified synchronization disciplines, e.g., all shared variables are protected by locks, transaction based methods [18, 19, 5] can be used to reduce the search space. These methods are orthogonal to our peephole reduction, and in principle transactions can be exploited along with our proposed reduction techniques. In addition to these conventional techniques, we present the following simplifications.

**Merging GIR Constraints.** If transition $t_1 \in trans_i$ is guarded independent with respect to all transitions $t_2 \in trans_j$, we do not need to add constraints separately for all $\langle t_1, t_2 \rangle$ pairs. Instead, we merge all these GIR constraints as

$$en_{t_1}(V^k) \wedge c_G(V^k) \rightarrow \neg(sel^k = j \wedge sel^{k+1} = i) \ .$$

As a simple case, this simplification can be applied when $t_1$ is a local transition (independent with all other threads). In this case, the effect captured is similar to that obtained from detecting transactions. However, the above rule is not restricted only to such simple cases. As a more general case, consider $N$ dining philosophers in which all transitions in one philosopher (thread) are visible to two adjacent philosophers (threads). There is no local transition *per se*. However, for any two non-adjacent philosophers, a transition $t_1$ in the $i$-th philosopher is always independent with all transitions in the $j$-th philosopher. Therefore the above simplification can be applied.

**Encoding Dependent Transitions.** For loosely coupled threads, the number of independent transition pairs are significantly larger than the number of *dependent transition pairs* (conflicts). In such cases, we can use an alternative encoding scheme. Instead of adding a constraint for every independent transition pair, we focus on dependent transition pairs. For threads $T_i$ and $T_j$ ($i < j$), we use $trans_i^C \subseteq trans_i$ and $trans_j^C \subseteq trans_j$ to denote the subsets of transitions that *may be guarded dependent*[4] with the other thread. By definition, $\forall t_1 \in (trans_i \setminus trans_i^C)$ and $\forall t_2 \in (trans_j \setminus trans_j^C)$,

[4] In $\langle t_1, t_2, c_G \rangle$, if $c_G$ is not constant true, then $t_1$ and $t_2$ *may be dependent*.

$t_1$ and $t_2$ are always independent. This is illustrated in Fig. 7. To encode the GIR constraints, first, we define $enable_{T_i}$ for thread $T_i$ as follows,

$$enable_{T_i} := \bigvee_{t \in (trans_i \setminus trans_i^C)} en_t \ .$$

Then, we summarize constraints for the *always independent* transition pairs.

$$enable_{T_i}(V^k) \wedge enable_{T_j}(V^k) \rightarrow \neg(sel^k = j \wedge sel^{k+1} = i) \ .$$

Finally, some transitions in $trans_i^C$ and $trans_j^C$ may still be independent from each other. For each $t_1 \in trans_i^C$ and $t_2 \in trans_j^C$, if $\langle t_1, t_2, c_G \rangle \in R_G$, we add the GIR constraint as in Section 3.2. As an example, if two threads are completely independent, only one constraint needs to be added to $SCH$.

## 5 Experiments

We have implemented the new methods in an SMT-based bounded model checker using the Yices SMT solver [4]. Yices is capable of deciding satisfiability formulae with a combination of theories including propositional logic, integer linear arithmetic, and arrays. We performed experiments with three variants of the peephole reduction, and a baseline BMC algorithm with no POR. The three variants represent different tradeoffs between the encoding overhead and the amount of achievable reduction. The first one is *static POR*, in which constraints are added only for statically independent transitions. The second one is *simple PPOR*, which adds constraints also for guarded independent transitions covered by GIR cases 1-3 (in Section 2.2). The third one is *full PPOR*, which adds constraints for all guarded independent transitions covered by GIR cases 1-4. Our experiments were conducted on a workstation with 2.8 GHz Xeon processor and 4GB memory running Red Hat Linux 7.2.

**Parameterized Examples.** The first set of examples are parameterized versions of *dining philosopher* and *indexer*. For dining philosopher, we used a version that guarantees the absence of deadlocks. Each philosopher (thread) has its own local state variables, and threads communicate through a shared array of chop-sticks. When accessing the global array, threads may have conflicts (data dependency). The first property (pa) we checked is whether all philosophers can eat simultaneously (the answer is no). The second property (pb) is whether it is possible to reach a state in which all philosophers have eaten at least once (the answer is yes). For the *indexer* example, we used the version from [6]. In this example, concurrent threads manipulate a shared hash table. Each thread needs to write four distinct values into the hash table. An atomic compare-and-swap instruction is used to check if a hash entry is available; if so, it writes the value; otherwise, the thread changes the hash key before retry. The property we checked is whether it is possible to reach a state in which all threads have completed. This example is interesting because there is no collision in accessing the hash table with up to 11 threads. However, such information cannot be detected by a static analysis.

For dining philosopher, with 2 threads we set the unrolling depths to 15,30,. . .,120, and compared the runtime of the four methods as well as the number of backtracks of

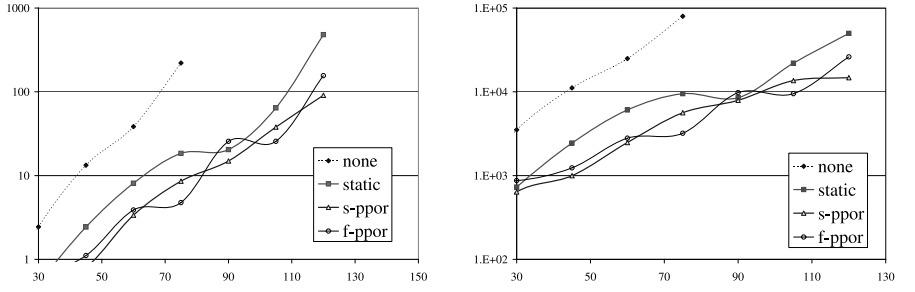



**Fig. 8.** Comparing runtime (left) and the number of backtracks in the SMT solver (right); performed on two philosophers with the property *pa*

**Table 1.** Comparing the performance of four symbolic partial order reduction techniques

| Test Program | | | Total CPU Time (s) | | | | #Conflicts (k) | | | | #Decisions (k) | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| name | steps | sat | none | static | s-ppor | f-ppor | none | static | s-ppor | f-ppor | none | static | s-ppor | f-ppor |
| phil2-pa | 15 | no | 0.3 | 0.1 | 0.1 | 0.1 | 0.5 | 0.1 | 0.1 | 0.9 | 1.1 | 0.6 | 0.6 | 0.4 |
| phil3-pa | 22 | no | 27 | 6 | 1.2 | 0.7 | 17 | 5 | 2 | 1 | 23 | 8 | 4 | 1 |
| phil4-pa | 29 | no | 69 | 50 | 26 | 28 | 39 | 28 | 13 | 13 | 54 | 41 | 21 | 20 |
| phil2-pb | 15 | yes | 0.1 | 0.1 | 0.1 | 0.1 | 0.1 | 0.1 | 0.1 | 0.1 | 0.5 | 0.6 | 0.4 | 0.3 |
| phil3-pb | 22 | yes | 1.7 | 0.8 | 0.4 | 1.4 | 2 | 1.1 | 0.4 | 1.0 | 3 | 2 | 1 | 3 |
| phil4-pb | 29 | yes | 19 | 17 | 1.5 | 2.9 | 12 | 9 | 1 | 2 | 17 | 15 | 4 | 5 |
| indexer2 | 10 | no | 0.3 | 0.2 | 0.1 | 0.1 | 0.3 | 0.3 | 0.1 | 0.1 | 1 | 1 | 1 | 1 |
| indexer3 | 15 | no | 31 | 23 | 0.4 | 0.3 | 14 | 14 | 0.4 | 0.3 | 183 | 183 | 1 | 1 |
| indexer4 | 20 | no | T/O | 1791 | 1.2 | 1.7 | - | 344 | 1 | 1 | - | 395 | 2 | 2 |
| indexer5 | 25 | no | T/O | T/O | 5.1 | 6.6 | - | - | 2 | 2 | - | - | 6 | 6 |
| indexer2 | 11 | yes | 3 | 2 | 0.4 | 0.7 | 1.5 | 1.5 | 0.3 | 0.6 | 54 | 54 | 15 | 33 |
| indexer3 | 16 | yes | 22 | 17 | 4 | 3 | 5 | 5 | 1 | 1 | 163 | 163 | 77 | 127 |
| indexer4 | 21 | yes | 179 | 177 | 12 | 6 | 283 | 283 | 3 | 2 | 432 | 432 | 181 | 139 |
| indexer5 | 26 | yes | T/O | T/O | 38 | 35 | - | - | 4 | 4 | - | - | 579 | 427 |

the SMT solver. The results are given in Fig. 8. The $x$-axis is the unrolling depths. The $y$-axis are the BMC runtime in seconds (left figure), and the number of backtracks (right figure). The $y$-axis is in logarithmic scale. The number of decisions of the SMT solver looks similar to the runtime curves; we omit it for brevity. These results show that both *simple PPOR* and *full PPOR* have shown significant performance improvement over *static*. Due to its larger encoding overhead, the runtime of *full PPOR* is less consistent and is sometimes inferior to *simple PPOR*.

We also set the number of threads to 2, 3, 4 for both dining philosopher and indexer examples and compared the four methods. In these experiments the BMC unrolling depths are chosen to be large enough (larger than the estimated reachable diameters [2]), so that the verification results are conclusive. The detailed results are given in Table 1. In Table 1, Columns 1-3 show the name of the examples, the number of BMC unrolling steps, and whether the property is true or not. Columns 4-7 report the runtime of the four methods. Columns 8-11 and Columns 12-15 report the number of backtracks and the number of decisions of the SMT solver.

**The Daisy Example.** The second set of examples come from a much larger concurrent program called Daisy. Daisy has been used before as a benchmark for verifying concurrent programs. The version we used is written in C and has been verified previously in [13]. The parsing and encoding of these examples were performed automatically. Note that in [13] there already exists a state-of-the-art symbolic POR method based on persistent sets, advanced static analysis techniques, and the exploitation of nested locks. Our peephole POR method was implemented on top of these techniques. The two properties we checked are data race conditions (both are reachable).

For comparison purposes, we implemented the peephole reduction on the same SAT-based BMC procedure as in [13]. Compared with the previous method, our peephole POR method can significantly reduce the BMC runtime in detecting these data races. In particular, for the first property, the new method was able to find a counterexample of length 132 and reduced the BMC runtime from 519 seconds to 374 seconds. For the second property, the new method was able to find a counterexample of length 136 and reduced the BMC runtime from 1540 seconds to 998 seconds.

## 6 Conclusions

We have presented a peephole partial order reduction method for model checking concurrent systems, based on a new notion of guarded independence relation between transitions. We have presented a concise symbolic encoding of local dynamically independent transition pairs which is well suited for using SMT/SAT solvers to find property violations. We have shown that the new peephole POR method can achieve significantly more reduction compared to other existing methods. For a system with two concurrent threads, our method guarantees the removal of all redundant interleavings. For future work, we plan to investigate additional techniques for simplifying the GIR constraints.

## References

[1] Alur, R., Brayton, R.K., Henzinger, T.A., Qadeer, S., Rajamani, S.K.: Partial-order reduction in symbolic state-space exploration. Formal Methods in System Design 18(2), 97–116 (2001)

[2] Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic model checking without BDDs. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, Springer, Heidelberg (1999)

[3] Cook, B., Kroening, D., Sharygina, N.: Symbolic model checking for asynchronous boolean programs. In: Godefroid, P. (ed.) SPIN 2005. LNCS, vol. 3639, pp. 75–90. Springer, Heidelberg (2005)

[4] Dutertre, B., de Moura, L.: A fast linear-arithmetic solver for dpll(t). In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 81–94. Springer, Heidelberg (2006)

[5] Dwyer, M.B., Hatcliff, J., Robby, Ranganath., V.P.: Exploiting object escape and locking information in partial-order reductions for concurrent object-oriented programs. Formal Methods in System Design 25(2-3), 199–240 (2004)

[6] Flanagan, C., Godefroid, P.: Dynamic partial-order reduction for model checking software. In: Principles of programming languages (POPL 2005), pp. 110–121 (2005)

[7] Godefroid, P.: Partial-Order Methods for the Verification of Concurrent Systems. LNCS, vol. 1032. Springer, Heidelberg (1996)

[8] Godefroid, P., Pirottin, D.: Refining dependencies improves partial-order verification methods. In: Courcoubetis, C. (ed.) CAV 1993. LNCS, vol. 697, pp. 438–449. Springer, Heidelberg (1993)
[9] Godefroid, P., Wolper, P.: Using partial orders for the efficient verification of deadlock freedom and safety properties. Formal Methods in System Design 2(2), 149–164 (1993)
[10] Grumberg, O., Lerda, F., Strichman, O., Theobald, M.: Proof-guided underapproximation-widening for multi-process systems. In: Principles of programming languages (POPL 2005), pp. 122–131 (2005)
[11] Gueta, G., Flanagan, C., Yahav, E., Sagiv, M.: Cartesian partial-order reduction. In: Bošnački, D., Edelkamp, S. (eds.) SPIN 2007. LNCS, vol. 4595, pp. 95–112. Springer, Heidelberg (2007)
[12] Ivančić, F., Shlyakhter, I., Gupta, A., Ganai, M.K., Kahlon, V., Wang, C., Yang, Z.: Model checking C program using F-Soft. In: International Conference on Computer Design, October 2005, pp. 297–308 (2005)
[13] Kahlon, V., Gupta, A., Sinha, N.: Symbolic model checking of concurrent programs using partial orders and on-the-fly transactions. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 286–299. Springer, Heidelberg (2006)
[14] Katz, S., Peled, D.: Defining conditional independence using collapses. Theor. Comput. Sci. 101(2), 337–359 (1992)
[15] Lerda, F., Sinha, N., Theobald, M.: Symbolic model checking of software. Electr. Notes Theor. Comput. Sci. 89(3) (2003)
[16] Mazurkiewicz, A.W.: Trace theory. In: Brauer, W., Reisig, W., Rozenberg, G. (eds.) APN 1986. LNCS, vol. 255, pp. 279–324. Springer, Heidelberg (1987)
[17] Peled, D.: All from one, one for all: on model checking using representatives. In: Courcoubetis, C. (ed.) CAV 1993. LNCS, vol. 697, pp. 409–423. Springer, Heidelberg (1993)
[18] Stoller, S.D.: Model-checking multi-threaded distributed java programs. International Journal on Software Tools for Technology Transfer 4(1), 71–91 (2002)
[19] Stoller, S.D., Cohen, E.: Optimistic synchronization-based state-space reduction. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 489–504. Springer, Heidelberg (2003)
[20] Valmari, A.: Stubborn sets for reduced state space generation. In: Rozenberg, G. (ed.) APN 1990. LNCS, vol. 483, pp. 491–515. Springer, Heidelberg (1991)