Partial Order Reduction in Symbolic State Space Exploration

R. Alur
Department of Computer and Information Science
University of Pennsylvania, Philadelphia, PA 19104
Email: alur@cis.upenn.edu

R.K. Brayton T.A. Henzinger S. Qadeer S.K. Rajamani
Department of Electrical Engineering and Computer Sciences
University of California at Berkeley, CA 94720
Email: {brayton, tah, shaz, sriramr}@eecs.berkeley.edu

September 7, 1999

Abstract

State space explosion is a fundamental obstacle in formal verification of designs and protocols. Several techniques for combating this problem have emerged in the past few years, among which two are significant: partial-order reductions and symbolic state space search. In asynchronous systems, interleavings of independent concurrent events are equivalent, and only a representative interleaving needs to be explored to verify local properties. Partial-order methods exploit this redundancy and visit only a subset of the reachable states. Symbolic techniques, on the other hand, capture the transition relation of a system and the set of reachable states as boolean functions. In many cases, these functions can be represented compactly using binary decision diagrams (BDDs). Traditionally, the two techniques have been practiced by two different schools—partial-order methods with enumerative depth-first search for the analysis of asynchronous network protocols, and symbolic breadth-first search for the analysis of synchronous hardware designs. We combine both approaches and develop a method for using partial-order reduction techniques in symbolic BDD-based invariant checking. We present theoretical results to prove the correctness of the method, and experimental results to demonstrate its efficacy.

1 Introduction

A system description consisting of parallel components has a state space that grows exponentially with the number of components. Commonly known as *state space explosion*, this is a fundamental problem in formal verification of systems like hardware designs, network protocols, and distributed algorithms. Techniques like model checking of temporal logic properties, which depend on state space exploration, have been limited in their use because of this problem. A variety of heuristics have been proposed to combat this problem. Two significant solutions that are supported by the existing tools are symbolic model checking and partial order reduction:

- In synchronous hardware designs, state space explosion manifests itself in the number of states growing exponentially with the number of state variables in the design. Symbolic model checking [BCM$^+$92, McM93] avoids explicit construction of the state space. The transition relation of the system and state sets are modeled as boolean functions and represented using Binary Decision Diagrams (BDDs). Model checking of temporal logic properties, then, reduces to symbolic fixpoint computation that uses BDD-based image computation as a primitive.

- In asynchronous systems consisting of a set of communicating concurrent processes (for example, network protocols and distributed algorithms) the behavior of the system can be modeled as the set of all interleavings of the events in individual processes. One source of state space explosion is due the $n!$ possible interleavings for n concurrent events. If the concurrent actions are independent then all interleavings are equivalent in that they lead to the same state. Partial order semantics described in [Maz89] for the composition of concurrent processes group equivalent interleavings of concurrent events together into *traces*. It has been shown that sometimes it suffices to explore just one representative interleaving from each trace for verifying temporal logic properties [Val90, Pel93, GW94]. Consequently, during state space exploration, in each state it suffices to search an *ample* subset of enabled transitions, rather than all of them, leading to significant reduction in the explored state space for some asynchronous protocols [HP94, God96].

In this work, we address the problem of *combining* partial order reduction methods with symbolic state space traversal. The method is particularly useful in cases where the number of states that have to be explored remains exponential in the size of the problem, even after partial order reduction. Consider, for example, a leader election protocol, where the initial states of the processes are unknown. The correctness criterion asserts that *irrespective of the initial states* of the processes, the desired outcome must be ensured. Since the number of initial states is exponential in the number of processes, explicit state space search would be forced to explore an exponential number of states, even with partial order reduction. Partial order reductions, however, give flexibility in the sets of states that need to be represented in symbolic search. Instead of computing images with the whole transition relation, it suffices to compute symbolically the ample set of successor states, or any set of successor states that contains the ample set. This flexibility adds a new dimension to heuristic techniques like modified search order [BCL91] and use of don't cares [RAP$^+$95], in symbolic state space exploration.

The following toy example illustrates the advantage of combining symbolic and partial-order methods. Consider n processes $P_1, P_2, \ldots, P_n$. Each process P_i increments a local variable x_i

independent of all the other processes, until $x_i = N$, after which x_i remains equal to N. Each of the x_i's is initialized to 0. We assume interleaving semantics for process composition, that is, with each transition, only one process updates its local variable. There are n transitions out of each global state in which $x_i \neq N$ for all i. For the purpose of comparing symbolic and enumerative approaches we assume, without loss of generality, that the variable ordering is $x_1, x_2, \ldots, x_n$. A naive enumerative algorithm has to explore $O(N^n)$ states. If we perform symbolic search using MDDs[1] (multi-valued decision diagrams) on this example, the set of reached states after k steps ($k \leq N$) is defined by the relation $\sum_{i=1}^{n} x_i \leq k$, whose MDD representation has size $\Theta(nk)$. Suppose we are interested in checking a local invariant, such as $x_3 \leq N$. For checking such an invariant, it is not necessary to explore all transitions out of a state. In fact, using partial-order reduction, it suffices to first increment x_1 until $x_1 = N$, then increment x_2 until $x_2 = N$, and so on. Thus, the number of explored states is reduced to $O(Nn)$. Both methods improve significantly on naive state space exploration.

Moreover, if symbolic search is performed on the partial-order reduced state space, the MDD representation of the set of reached states after any number of steps has size $\Theta(n)$, which is an improvement over either method in isolation. This improvement becomes even more significant if the variables have nondeterministic initial values. Suppose that each x_i is initialized nondeterministically to either 0 or 1. An enumerative algorithm has to visit 2^n initial states, even with partial-order reduction.[2] In symbolic search, the set of reached states after k steps ($k \leq N$) is defined by the relation $\sum_{i=1}^{n} max(x_i - 1, 0) \leq k$. It can be shown that the size of the MDD representing the above relation is $\Omega(n^2 + nk)$. However, for symbolic state space traversal using the partial-order reduced transition relation, the MDD representing the set of reached states after any number of steps is only $\Theta(n)$. Similar improvements in search efficiency are obtained in two examples in Section 3—a leader-election protocol in a unidirectional ring and an asynchronous tree arbiter circuit.

In [Val90, Pel93, GW94], partial-order reductions are obtained by a modified depth first search of the state space. In Section 2, we show that correct partial-order reductions can be obtained on any search technique that has a certain property. Breadth-first search is shown to have this property. Consequently, partial-order reductions can be applied to symbolic search (which is inherently breadth-first). A similar generalization was also obtained by [CP96] in the context of mechanically verifying the partial order reduction techniques implemented in the model checker SPIN [HP94]. Partial-order methods explore only a subset of the enabled transitions at each state, called an *ample set*. For the automatic implementation of our method, ample sets have to be computed automatically in a symbolic setting.

[1]MDDs are a generalization of BDDs in which the variables can take on values from any finite set instead of just {0,1}.

[2]In this particular example, since the processes are initialized independently and the behavior of the system from different initial states is identical, the system description could be modified so that enumerative search with partial-order reduction is efficient. But such a simplification is not possible in the general case.

2 Partial Order Reduction for Breadth First Search

2.1 Preliminaries

A *labeled transition graph* is a 5-tuple $G = \langle V, S, \hat{S}, \Gamma, \rightarrow \rangle$ with the following components:

- A finite set V of binary variables.
- The set $S = 2^V$ of all possible valuations to the variables, called the state set.
- A set of initial states $\hat{S} \subseteq S$.
- A finite set Γ of *actions*.
- A transition relation $\rightarrow \ \subseteq \ S \times \Gamma \times S$, with the restriction that if (s, α, t) and (s, α, t') are in $\rightarrow$, then $t = t'$.

We write $s \stackrel{\alpha}{\rightarrow} t$ if $(s, \alpha, t) \in \ \rightarrow$, and $s \rightarrow t$ if there exists an action α, such that $s \stackrel{\alpha}{\rightarrow} t$. The set of all actions enabled in state s is defined as $enabled(s) = \{\alpha \mid \exists t \ s \stackrel{\alpha}{\rightarrow} t\}$. An action sequence $\overline{\alpha} = \alpha_1\alpha_2 \ldots \alpha_m$ in Γ^* is *G-enabled* in state s if there exist states $s_1, \ldots, s_{m-1}, t$ such that $s \stackrel{\alpha_1}{\rightarrow} s_1 \stackrel{\alpha_2}{\rightarrow} \ldots \stackrel{\alpha_{m-1}}{\rightarrow} s_{m-1} \stackrel{\alpha_m}{\rightarrow} t$. The state t is denoted by $succ(s, \overline{\alpha})$ and we write $s \stackrel{\overline{\alpha}}{\Rightarrow} t$. The set of all reachable states of G is defined as $Reach(G) = \{s \mid \exists \overline{\alpha} \in \Gamma^*, \hat{s} \in \hat{S} \text{ such that } s = succ(\hat{s}, \overline{\alpha})\}$. The first action of an action sequence $\overline{\beta}$ is denoted by β_1.

The algorithm to obtain a partial-order reduction performs a selective search. While exploring the successors of a state s, instead of considering all the actions enabled in s, the algorithm chooses only a subset of the enabled transitions, and thus, constructs a subgraph of the original graph. Let $G = \langle V, S, \hat{S}, \Gamma, \rightarrow \rangle$ be a labeled transition graph. A graph $G' = \langle V, S', \hat{S}, \Gamma, \rightarrow' \rangle$ is a subgraph of G if $S' \subseteq S$ and $\rightarrow' \ \subseteq \ \rightarrow$. The selective search will be specified by a function from states to subsets of actions. For a function Δ from S to 2^Γ, the subgraph $G_\Delta = \langle V, S_\Delta, \hat{S}, \Gamma, \rightarrow_\Delta \rangle$ of G is a *reduction* of G w.r.t. the function Δ, if for every state $s \in S_\Delta$, $s \rightarrow_\Delta t$ iff for some $\alpha \in \Delta(s)$, $t = succ(s, \alpha)$. We write $s \stackrel{\overline{\alpha}}{\Rightarrow}_\Delta t$ if $\overline{\alpha}$ is G_Δ-enabled in state s and $succ(s, \overline{\alpha})$ is t. In the remainder of this section, we develop requirements on Δ so that the reduction G_Δ can be used for invariant verification.

2.2 Partial order equivalence

We proceed to define an equivalence relation over the action sequences of a labeled transition graph so that, when solving a reachability problem , it suffices to explore only one representative from each equivalence class. This equivalence is defined using the notion of independence between actions.

Let α and β be two different actions of G. The two actions α and β are *independent*, written $\alpha \sim \beta$, if for all states $s \in S$, (1) if α is enabled in s then, β is enabled in s iff β is enabled in $succ(s, \alpha)$, (2) if β is enabled in s then, α is enabled in s iff α is enabled in $succ(s, \beta)$, (3) if both α and β are enabled in s then $succ(s, \alpha\beta) = succ(s, \beta\alpha)$. Intuitively, two actions are independent if they neither enable nor disable each other, and the order in which the two actions are executed does not affect the final state reached. Figure 1 shows two independent actions α and β. For every labeled transition graph G, the independence relation $\sim$ on the actions of G is irreflexive and symmetric.

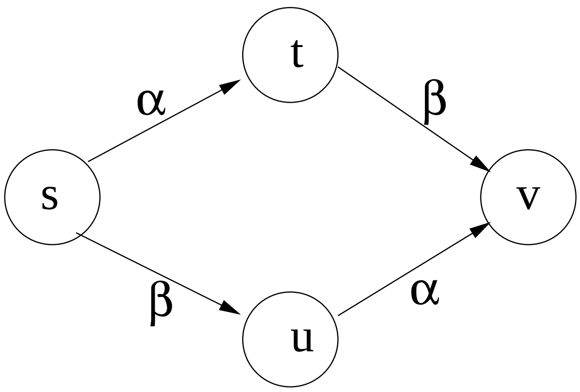

Figure 1: Independent actions

For two action sequences $\overline{\gamma}$ and $\overline{\delta}$ of a labeled transition graph G, define $\overline{\gamma} \equiv \overline{\delta}$ if $\overline{\gamma} = \overline{\gamma_1}\alpha\beta\overline{\gamma_2}$ and $\overline{\delta} = \overline{\delta_1}\beta\alpha\overline{\delta_2}$, with $\alpha \sim \beta$. The *partial order equivalence* $\equiv^*$ on the action sequences of G is the reflexive-transitive closure of $\equiv$. Intuitively, two action sequences are partial-order equivalent if one can be obtained from the other by repeatedly commuting adjacent independent actions. The set of all action sequences partial order equivalent to the action sequence $\overline{\alpha}$ is represented by $[\overline{\alpha}]$. Such an equivalence class is called a Mazurckiewicz trace [Maz89].

2.3 Partial order reduction

2.3.1 Persistent and ample functions

During the selective state space search, only a subset of the actions enabled in a state are explored. In this section, we derive conditions on this subset so that the selective search method is sound and complete for local invariants.

The actions in the selected subset should be independent, not only of all the remaining actions enabled in state s, but also independent of all actions enabled in a state reachable from s by executing other actions. Such a subset of actions is called persistent. Formally, a function Δ from S to 2^{Γ} is *persistent* if for every state s of G the following holds: for all actions $\alpha \in \Delta(s)$, (1) α is enabled in s, and (2) if there is an action sequence $\overline{\beta} = \beta_1\beta_2 \ldots \beta_n \in (\Gamma \backslash \Delta(s))^*$ that is G-enabled in s, then α is independent of every action β_i in $\overline{\beta}$. We say that $\Delta(s)$ is a *persistent set* of actions at s.

Lemma 2.1 *[God96] Let s be a state in G and $\overline{\alpha}$ an action sequence G-enabled in s. Let Δ be a persistent function such that $\Delta(s)$ is non-empty. If $\beta_1 \notin \Delta(s)$ for all action sequences $\overline{\beta} \in [\overline{\alpha}]$, then all actions in $\Delta(s)$ are independent of all actions in $\overline{\alpha}$.*

Proof. The proof is by contradiction. Let $\overline{\alpha} = \alpha_1\alpha_2 \ldots \alpha_n$. Suppose there exist transitions in α that are dependent on some transition in $\Delta(s)$. Let α_k be the first transition in $\overline{\alpha}$ which is not independent of all transitions in $\Delta(s)$. Then $\alpha_k \in \Delta(s)$ since α_k cannot be dependent on any transition in $\Delta(s)$ by the definition of persistent function. Since $\alpha_k \in \Delta(s)$, the action sequence $\alpha_k\alpha_1 \ldots \alpha_{k-1}\alpha_{k+1} \ldots \alpha_n \in [\overline{\alpha}]$. Therefore $\alpha_k = \beta_1$ for some $\overline{\beta} \in \overline{\alpha}$. This contradicts the assumption that $\beta_1 \notin \Delta(s)$. ■

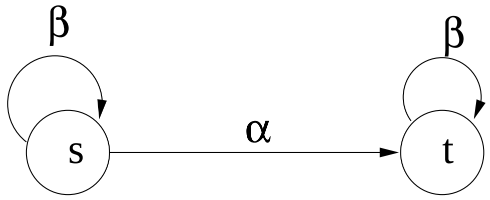

Figure 2: Action α is delayed forever

We need to make sure that a selective search based on a persistent mapping does not delay an action forever. This could happen, for instance, if an action α is enabled everywhere in a cycle containing a state s. If s is the first state in the cycle visited by the selective search, then α could be delayed until s is visited again, which is tantamount to delaying it forever. To avoid this, all enabled actions in a state are explored if the persistent set does not lead to a *new* state. An example is the state s and action α in Figure 2. Notice that $\{\beta\}$ is a persistent set in state s.

Definition 2.1 Given a persistent function Δ, a mapping Ψ from states of the reduction G_Δ to $\mathbb{N}$, the set of natural numbers, is said to be a <u>*witness*</u> for Δ, if for all states s, if $\Delta(s) \neq \mathit{enabled}(s)$, then there exists an action $\alpha \in \Delta(s)$ such that $\Psi(\mathit{succ}(s, \alpha)) < \Psi(s)$.

The well-foundedness of Ψ is then used to ensure that an action is not ignored everywhere in a cycle. A *partial order reduction* of G is a reduction of G with respect to a persistent function that has a witness.

Lemma 2.2 *[God96] Let Δ be a persistent function and G_Δ be a partial order reduction of G with respect to Δ. Let s be a state in G_Δ and $\overline{\alpha}$ an action sequence G-enabled in state s. Then, there exists an action sequence $\overline{\alpha}'$ every action of which is independent of all actions in $\overline{\alpha}$ such that $s \stackrel{\overline{\alpha}'\beta_1}{\Longrightarrow}_\Delta s'$ for some $\overline{\beta} \in [\overline{\alpha}]$ and state s'.*

Proof. Since G_Δ is a partial order reduction of G, the persistent function Δ has a witness, say Ψ. The proof is by induction on $\Psi(s)$.

Base case: $\Psi(s) = 1$. Since Ψ is a witness, we know that for any t $\Psi(t) \geq 1 = \Psi(s)$. Hence, we must have $\Delta(s) = \mathit{enabled}(s)$. We have $\overline{\alpha}' = \epsilon$.

Inductive case: Assume the lemma true for all $1 \leq \Psi(s) \leq n$. Let $\Psi(s) = n + 1$. If $\beta_1 \in \Delta(s)$ for some $\overline{\beta} \in [\overline{\alpha}]$, then $\overline{\alpha}' = \epsilon$. Suppose $\beta_1 \notin \Delta(s)$ for all $\overline{\beta} \in [\overline{\alpha}]$. Then, clearly $\Delta(s) \neq \mathit{enabled}(s)$. Since Ψ is a witness, there exists $\gamma \in \Delta(s)$ such that $s \stackrel{\gamma}{\rightarrow}_\Delta t$ and $\Psi(t) < \Psi(s)$. Since Δ is persistent, by Lemma 2.1, all actions in $\overline{\alpha}$ are independent of all actions in $\Delta(s)$. Therefore, $\overline{\alpha}$ is G-enabled in state t. Applying the inductive hypothesis to t, there exists a state s' such that $t \stackrel{\overline{\alpha}''\beta_1}{\Longrightarrow}_\Delta s'$ in G_Δ, for some $\overline{\beta} \in [\overline{\alpha}]$ and all actions in $\overline{\alpha}''$ are independent of all actions in $\overline{\alpha}$ in G. Noting that γ is independent of all actions in $\overline{\alpha}$, the action sequence $\overline{\alpha}' = \gamma\overline{\alpha}''$ is the desired sequence such that $s \stackrel{\overline{\alpha}'\beta_1}{\Longrightarrow}_\Delta s'$. ■

Definition 2.2 Let $\overline{\alpha}$ and $\overline{\beta}$ be action sequences. We say $\overline{\beta}$ <u>*subsumes*</u> $\overline{\alpha}$, if there exists an action sequence $\overline{\gamma}$ such that $\overline{\alpha\gamma} \equiv^* \overline{\beta}$.

The relationship between a graph and its partial order reduction is captured by the following theorem, which states that every action sequence enabled in an initial state of the original graph is subsumed by some sequence enabled in the reduction.

Theorem 2.1 *[God96] Let s be a state in a partial order reduction G_Δ of a labeled transition graph G, and let $\overline{\alpha}$ be an action sequence G-enabled in s. There exists an action sequence $\overline{\beta}$ that is G_Δ-enabled in s such that $\overline{\alpha}$ is subsumed by $\overline{\beta}$.*

Proof. The proof is by induction on the length of $\overline{\alpha}$. The base case $|\overline{\alpha}| = 0$ is trivial.

Inductive case: Assume the theorem holds for all $0 \leq |\overline{\alpha}| \leq n$. Let $|\overline{\alpha}| = n+1$. By Lemma 2.2, we know $s \overset{\overline{\alpha}'\beta_1}{\Longrightarrow}_\Delta s'$ for some $\overline{\beta} \in [\overline{\alpha}]$, state s' and action sequence $\overline{\alpha}'$ such that all actions in $\overline{\alpha}'$ are independent of all actions in $\overline{\alpha}$. This implies that $\overline{\alpha}$ is G-enabled in all intermediate states reached from s by $\overline{\alpha}'$ including $succ(s, \overline{\alpha}')$. Therefore, a sequence $\overline{\gamma}$ is G-enabled in s' such that $\beta_1\overline{\gamma} \equiv \overline{\alpha}$. Applying the inductive hypothesis to $\overline{\gamma}$ ($|\overline{\gamma}| = n$), there is an action sequence $\overline{\alpha}''$ which is G_Δ-enabled in s' such that $\overline{\gamma}$ is subsumed by $\overline{\alpha}''$. Thus, we have the sequence $\overline{\alpha}'\beta_1\overline{\alpha}''$ G_Δ-enabled in s such that $\overline{\alpha}$ is subsumed by it. ■

Thus, to compute partial order reductions, we need persistent functions that can be shown to have a witness. For this purpose, we use *ample* functions that take into account the set of states visited so far. An ample function is defined with respect to a history function τ from S to 2^S that keeps track of the history of the state space search. Using ample functions with respect to search history is a constructive method for generating a persistent function with a witness.

Definition 2.3 A *history function* τ is a map from S to 2^S. A persistent function Δ is said to be *ample* with respect to a history function τ if for every state s, either $\Delta(s) = enabled(s)$ or there is an action $\alpha \in \Delta(s)$ such that $succ(s, \alpha) \notin \tau(s)$.

Thus, an ample function insists on exploring all the enabled transitions or visiting some new (i.e. not in the current history) state. Later, we will define a history function τ_B and a numbering Ψ_B for the breadth-first search so that Ψ_B is a witness for any ample function with respect to τ_B. If Δ is an ample function with respect to the history τ, then we say that $\Delta(s)$ is an ample set at state s.

2.3.2 Verification using partial order reduction

Partial order reductions are sound and complete for checking local properties. We formalize local properties as *independence preserving regions.* In particular, we show if there is an action sequence that originates at an initial state and leads to an independence preserving region, there exists some action sequence in any partial order reduction that leads to the same region.

Definition 2.4 A set $\sigma \subseteq S$ of states of a graph G is an *independence preserving region* if for all states s in the set of initial states, whenever $succ(s, \overline{\alpha}) \in \sigma$ and $\overline{\beta}$ subsumes $\overline{\alpha}$, there exists a prefix $\overline{\gamma}$ of $\overline{\beta}$ such that $succ(s, \overline{\gamma}) \in \sigma$.

We say that a set of states $\sigma^I \subseteq S$ is an *independence preserving invariant* if $S \backslash \sigma^I$ is an independence preserving region. We say that an invariant σ_I is *satisfied* by G if $Reach(G) \subseteq \sigma_I$. It

```
module Inc
  interface x,y:ℕ
  atom controls x, y
  init
      []true -> x' := 0; y' := 0
  update
     a    [] true  ->
     a_x  [] x<10  -> x' := x+1
     a_y  [] y<10  -> y' := y+1
  endatom
endmodule
```

Figure 3: Example of independence preserving region

follows that to verify that a given graph G satisfies an invariant σ^I, we can check whether some partial order reduction of G satisfies the invariant σ^I, provided σ^I is an independence preserving invariant. To determine in practice whether an invariant is independence preserving, one can use the sufficient condition of locality. A predicate p on the variables of the transition system is said to be *local* if for all actions α and β that modify a variable occurring in p, we have that α is dependent with β. It can be shown that if a predicate p is local then the set of states where p is true, is an independence preserving region. In Figure 3, the module Inc has three actions —a, a_x and a_y— that are all independent of each other. The action a_x modifies variable x only, action a_y modifies variable y only, and action a does not modify either x or y. The predicates $x < 5$ and $y < 10$ are local and consequently independence preserving.

2.4 Partial order reduction with breadth first search

The algorithm for constructing a partial order reduction during BFS is a variation of the standard BFS algorithm. The idea is to use the current set of reached states as the history function. Further, since our aim is to do the BFS symbolically, we explore *all* the states reachable in one step by the set of ample actions in the current frontier simultaneously. More formally, let R_k be the set of reached states after the kth reachability step ($R_0 = \hat{S}$). Then, if $s \in R_{k+1} \backslash R_k$, define $\tau_B(s) = R_{k+1}$. The pseudo-code for the algorithm is given in Figure 4.

There are two types of states in the reduced state graph, which we denote by G_B. Some states are never a part of any frontier. Other states are a part of some frontier exactly once. We will define a mapping Ψ_B from states of G_B to $\mathbb{N}$ as follows: (1) if s is never a part of any frontier then $\Psi_B(s) = 1$; (2) if s is a part of i^{th} frontier and the total number of frontiers is N, then $\Psi_B(s) = N - i + 2$. Figure 5 illustrates the mapping for a reduced state graph with 4 frontiers. The shaded areas are the frontiers and the unshaded areas are the interior states.

To prove the correctness of the reduction, we need to establish that as long as we choose an ample set at every step with respect to the history τ_B, the numbering Ψ_B is a witness:

σ_{new} := set of initial states
Repeat
/* σ is the current set of reached states */
$\sigma := \sigma_{new}$
Frontier $\mathcal{F} := \{s \in \sigma \mid \exists t \notin \sigma.\ s \rightarrow t\}$
For every $s \in \mathcal{F}$, let $\Delta(s)$ be an ample set at s with respect to the history τ_B
$\sigma_{next} := \cup_{s \in \mathcal{F}} \{t \mid \exists \alpha \in \Delta(s).\ s \xrightarrow{\alpha} t\}$
$\sigma_{new} := \sigma \cup \sigma_{next}$
Until $(\sigma = \sigma_{new})$
Return

Figure 4: Modified breadth first search algorithm

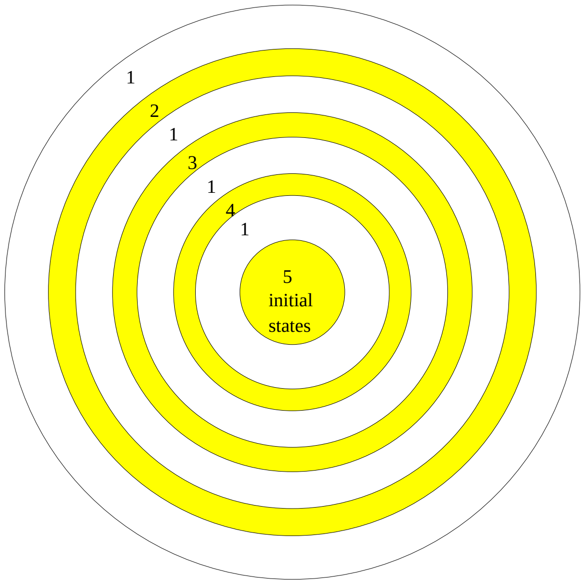

Figure 5: Ψ_B for states during BFS expansion

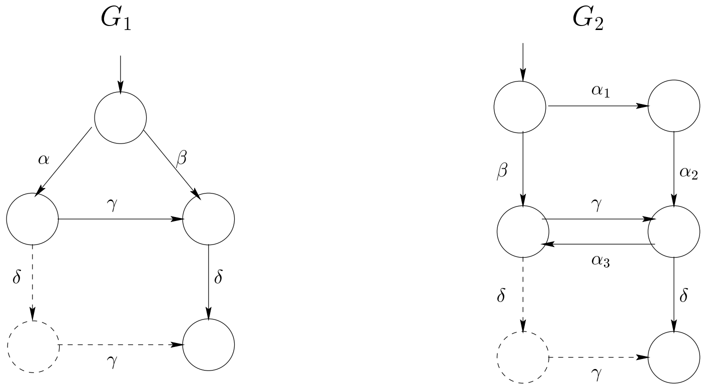

Figure 6: Comparison between enumerative breadth-first and depth-first search

Lemma 2.3 *Let Δ be an ample function with respect to the history τ_B. If $\Delta(s) \neq enabled(s)$ then for some $\alpha \in \Delta(s)$, $\Psi_B(succ(s,\alpha)) < \Psi_B(s)$.*

Proof. Since $\Delta(s) \neq enabled(s)$, there is an action $\alpha \in \Delta(s)$ such that $succ(s,\alpha) \notin \tau_B(s)$. This means that s is a part of some frontier, say i. Therefore, $succ(s,\alpha)$ is either a part of the $(i+1)^{th}$ frontier or not a part of any frontier. In either case, the definition of Ψ_B requires that $\Psi_B(succ(s,\alpha)) < \Psi_B(s)$. ∎

Theorem 2.2 *Let G_B be the reduced graph obtained by the modified BFS algorithm. G_B is a partial order reduction of G.*

Proof. The ample function Δ used in the construction of the reduced graph G_B is a persistent function by definition. From Lemma 2.3, Ψ_B is a witness for Δ. Hence, G_B is a partial order reduction of G. ∎

We note that depth first search can also be used to construct a partial order reduction. The set of states in the stack can be used as the history function for this [God96]. If the criterion for comparison is number of states visited, there is no clear winner between the enumerative breadth-first and depth-first algorithms. We compare the two in Figure 6. In both transition graphs G_1 and G_2, only transitions γ and δ are independent of each other. In the transition graph G_1, a breadth-first search explores every state but a depth-first search explores only the states and transitions drawn in solid line. In the transition graph G_2, a depth-first search that explores the transition α_1 before β will explore every state but a breath-first search explores only the states and transitions drawn in solid line.

3 Experiments

We consider two examples for experiments—a protocol for electing a leader among a set of processes and an asynchronous tree arbiter circuit. We describe below how we use scheduler modules to build Verilog models for the original system as well as the reduced system obtained by considering only those actions from a state that form an ample set. To each Verilog module i, we added a boolean variable called $enable_i$ such that all transitions of the module are disabled if $enable_i$ is set to false. We then add a scheduler that controls all these variables. We analyzed the examples ourselves and came up with ample sets. In the experiments, we compare an "ample scheduler" that nondeterministically chooses one among a subset of enabled processes (corresponding to the ample set) at each state, to a completely nondeterministic scheduler that may enable any process at any time. The symbolic reachability computation was performed with VIS [BHSV$^+$96]. There are a number of algorithms available in the literature for computing ample sets. Although it is possible to compute ample sets symbolically, the computations require the identification of actions in the representation of transition systems, which VIS does not support at present. In the appendix, we take an example algorithm and demonstrate how it can be implemented symbolically. The purpose of our experiments is to investigate if savings in BDD sizes could be obtained, if such ample sets were available, and the answer is in the affirmative.

3.1 Leader-election protocol

We consider the leader-election protocol described in [DKR82]. Extensive reduction in reachable state space sizes for this problem is reported by [HP94], using partial-order reductions with enumerative depth-first search of the state space.

There are N processes `P0`, `P1`, ... connected in a circular ring, with a finite FIFO unidirectional queue between adjacent processes. The module `node` in Figure 7 describes the behavior of one process. Each process has a unique id associated with it that is constant through the operation of the protocol. Only those initial states are valid where the ids of the processors are distinct numbers between 0 and $N-1$. The module `chan` is a unidirectional FIFO queue (its description has been omitted from Figure 7). A process receives messages from the queue immediately before it in the ring and can send messages to the queue immediately after it. Each process has a variable called `active` associated with it. Initially `active` is `true` for all processes. The objective of the protocol is to eventually elect a unique leader process with `active` being `true`. The messages put on the FIFO queues contain id's of processes currently active. Initially, each process sends its id to the next process and receives the id of the preceding process. Every process remembers the maximum id it has seen so far. As a result of comparing received process id's to the maximum id, a process may deactivate itself. Finally, only one process should remain active. The pseudo-code for the protocol with five processes is given in Figure 7. To check the property that finally a unique leader exists using an independence-preserving invariant, we introduce a variable called `flag` for each process. As soon as a process knows that it is the leader, it asserts `flag` and `flag` does not change thereafter. The independence-preserving invariant to be checked is that at all times at most one `flag` variable is true.[3]

A process is enabled if there is at least one message in its input queue and at least one empty slot in its output queue. In every state, every enabled process has precisely one enabled action. For each state, we consider the singleton set consisting of the action that is owned by the process with least index among the enabled processes. This set is a non-empty persistent set, because the

[3] This, of course, is not a complete specification of the protocol. A liveness property stating that eventually a process does become the leader is also needed.

```
rtype = record of (valid:bool, a:{first, second}, b:(0..2));

module node
  external in:rtype; full:bool
  interface out:rtype; read:event
  private active:bool; id,maxi,neighborR:(0..2)
  atom controls id reads id
  update
      []true -> id':=id
  endatom

  lazy atom controls out, maxi, neighborR, active, read
        reads in, active, maxi, neighborR, read, full awaits id
  init
    []true -> maxi':=id'; active':=true; out':=(true,first,id')
  update
    [](in.valid & in.a=first)  &            -> read!; neighborR':=in.b;
      active  & ¬full  & in.b≠maxi             out':=(true,second,in.b)
    [](in.valid & in.a=first)  &            -> read!;
      ¬active  & ¬full                         out':=(true,first,in.b)
    [](in.valid & in.a=second)  &           -> read!; maxi':=neighborR;
      active  & ¬full  &                       out':=(true,first,neighborR)
      (in.b<neighborR  & maxi <neighborR)
    [](in.valid & in.a=second)  &           -> read!; active':=false
      active  &                                out'.valid:=false
      ¬(in.b<neighborR  & maxi <neighborR)
    [](in.valid & in.a=second)  &           -> read!;
      ¬active  & ¬full                         out':=(true,second,in.b)
    []true                                  -> out'.valid:=false
  endatom
endmodule

/* cin,cout:array (0..2) of (0..2); qread:array (0..2) of event;
qfull:array (0..2) of bool */
P0 := node [in,read,out,full:=cout[2],qread[2],cin[0],qfull[0]];
Q0 := chan [in,full,out,read:=cin[0],qfull[0],cout[0],qread[0]];
P1 := node [in,read,out,full:=cout[0],qread[0],cin[1],qfull[1]];
Q1 := chan [in,full,out,read:=cin[1],qfull[1],cout[1],qread[1]];
P2 := node [in,read,out,full:=cout[1],qread[1],cin[2],qfull[2]];
Q2 := chan [in,full,out,read:=cin[2],qfull[2],cout[2],qread[2]];
System := P0 || Q0 || P1 || Q1 || P2 || Q2
```

Figure 7: Pseudo-code for leader election protocol

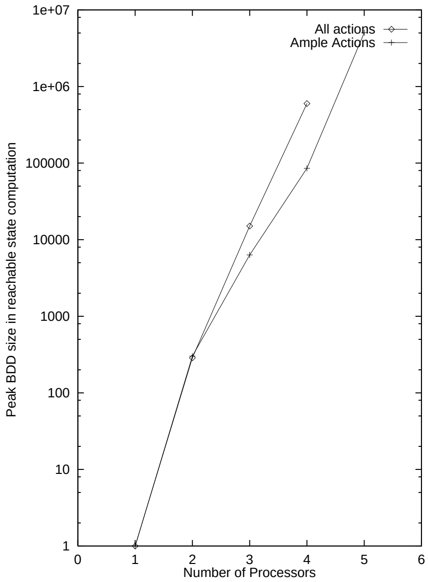

Figure 8: Leader-election protocol: peak BDD sizes needed for reachability analysis

protocol has the property that if multiple processes are enabled at the same time, the order in which the processes execute actions does not change the eventual outcome. The transition graph of the protocol is acyclic and hence every persistent set is ample.

Exponential partial-order reductions on this example are reported in [HP94] for a ring of 5 processes. There, however, the ids of the processes are assumed to be initialized to 0, 1, 2, 3, and 4 in circular order around the ring. Though the protocol is verified for this initial state, it is not verified for other initial states. For instance, if the id's are 0, 4, 1, 3, and 2, then the behavior of the protocol is very different. The former initialization requires a total of 15 messages passed around the ring before a leader is elected, whereas the latter requires 25 messages. In fact, there are certain lines of code in the protocol that are never exercised in the former case but are exercised in the latter. Our method verifies the correctness claim as stated in [DKR82], for *all* initial states.

Figure 8 shows the growth of peak BDD size during reachability analysis versus the number of processes. Two curves are shown—one for the entire transition relation and another for the ample transition relation. Note that reachability analysis with the entire transition relation runs out of memory for 5 processes. With partial-order reduction, in the case of 5 processes, VIS reports 3.76×10^{15} states visited. Thus, in this example, partial-order reduction coupled with symbolic search performs better than either method in isolation.

3.2 Asynchronous tree arbiter

The tree arbiter is an asynchronous circuit which solves the mutual-exclusion problem by building a tree of arbiter cells. The circuit does the arbitration in the form of an elimination tournament. An arbiter cell arbitrates between its two children. The leaves in the tree are processors that asynchronously demand access to a common resource. The n users at the lowest level in the tree

```
module ArbiterCell
  private pc:{outCS,reqCS,inCS}
  external ur1,ur2,sa:bool
  interface ua1,ua2,sr:bool
  atom controls pc,ua1,ua2,sr reads pc,ur1,ur2,ua1,ua2,sa
  init
      []true -> pc':=outCS; ua1':=false; ua2':=false; sr':=false
  update
    a   []pc=outCS & (ur1 | ur2)                 -> pc':=reqCS; sr':=true
    b1  []pc=reqCS & sa & ur1                    -> pc':=inCS; ua1':=true
    b2  []pc=reqCS & sa & ur2                    -> pc':=inCS; ua2':= true
    c   []pc=inCS & (ua1 & ~ur1 | ua2 & ~ur2)    -> pc':=relCS; sr':=false
    d   []pc=relCS & ~sa                         -> pc':=outCS;
                                                    ua1':=false; ua2':=false
  endatom
endmodule
module Processor
  private pc:{outCS,reqCS,inCS}
  external ack:bool
  interface req:bool
  atom controls pc,req reads pc,ack
  init
      []true -> pc':=outCS; req':=false
  update
    α'  []pc=outCS          ->
    α   []pc=outCS          -> pc':=reqCS; req':=true
    β   []pc=reqCS & ack    -> pc':=inCS
    γ'  []pc=inCS           ->
    γ   []pc=inCS           -> pc':=relCS; req':=false
    δ   []pc=relCS & ~ack   -> pc':=outCS
  endatom
endmodule
module Buffer
  external in:bool
  interface out:bool
  atom controls out awaits in
  init update
      []true -> out':=in'
  endatom
endmodule
Processor1:= Processor[req,ack:=ur1,ua1]
Processor2:= Processor[req,ack:=ur2,ua2]
System:= Processor1|| Processor2|| ArbiterCell || Buffer[out,in:=sa,sr]
```

Figure 9: Pseudo-code for tree arbiter

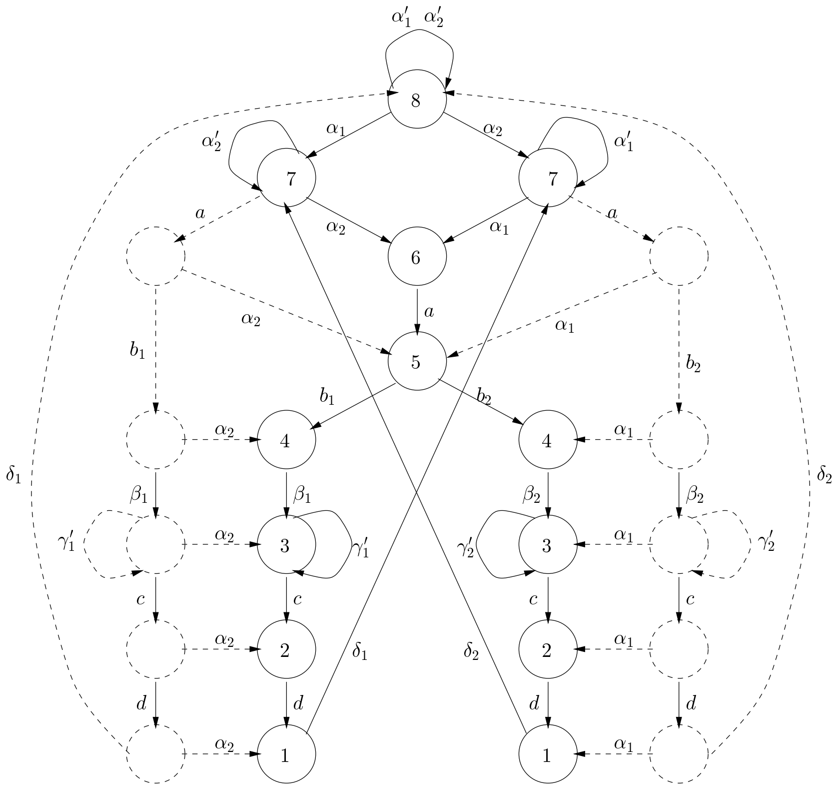

Figure 10: State transition graph of `System` in Figure 9

are arbitrated by $\frac{n}{2}$ cells. The winners at this level are arbitrated at the next level and so on. If both children of an arbiter cell request, one of them is chosen nondeterministically. The requests propagate upward in the tree until the root cell is reached. It grants the resource to at most one of its children and the grant propagates downward to one of the processors. Similarly, when the resource is released by a processor, the request for release goes all the way up and is acknowledged all the way down. We model the arbiter cells as interacting asynchronously with each other. Four-phase signaling is used for communication between the arbiter cells. Figure 9 shows the pseudo-code for 2 users and 1 arbiter cell. The actions labeling the different guarded commands are written beside them. The two instantiations of actions $\alpha', \alpha, \beta, \gamma', \gamma, \delta$ in `Processor1` and `Processor2` are obtained by subscripting the actions by the processor index.

We label all processes (processors and arbiter cells) by levels, starting at the leaves: at the lowest level, label the processors from left to right with the leftmost processor having the least label; then label the first level of arbiters from left to right; etc. Given such a labeling, it can be shown that at each state, the set containing all enabled transitions of the enabled process with the least label is an ample set. We illustrate this by example on the arbiter circuit with 2 processors and 1 arbiter cell from Figure 9. The labeled transition graph of `System` is given in Figure 10. The states and transitions explored by the selective search according to the ample set described above are shown in

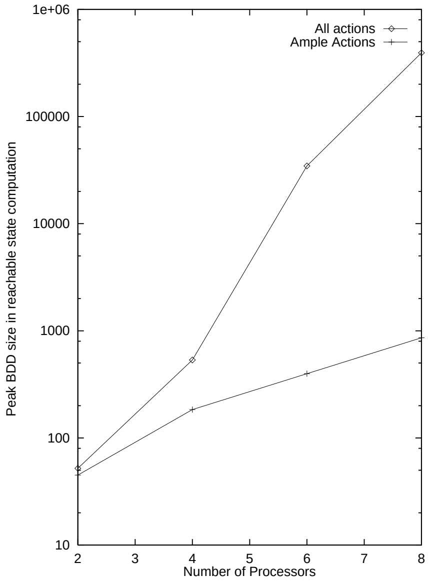

Figure 11: Tree arbiter: peak BDD sizes needed for reachability analysis

solid line; the unexplored states and transitions are shown in dashed line. It is not too hard to see that the set of actions explored from any state in the reduced graph is a persistent set. Moreover, the numbers labeling the states in the reduced graph form a witness for the persistent sets. Hence, we have a correct partial-order reduction.

Figure 11 shows the BDD sizes during reachability analysis, for the entire transition relation and the ample transition relation (8 processors induce a tree with three levels of arbitration). Note the log scale on the vertical axis of the plot. The memory requirement for the entire transition relation grows exponentially with the number of processors, whereas the memory for the ample transition relation grows polynomially. The difference in memory consumption also manifests itself in run time: for 8 processors, reachability analysis with the entire transition relation requires 1755 seconds, whereas the ample transition relation requires less than 1 second to complete.

4 Conclusions

In this work, we identified the need for coupling partial order reduction with symbolic methods. We presented a modified breadth first state-space search algorithm for verifying a class of local properties called independence preserving invariants. We used this algorithm to do reachability symbolically and demonstrated exponential reduction in the size of the BDDs needed for the computation on two examples.

While partial order reduction can decrease the number of states, it does not guarantee a reduction in the BDD sizes directly. At each reachability step, partial order reduction provides flexibility in the set of states that need to be explored. As a preliminary exploration of this flexibility, we performed the reachability computation in a different way. If σ is the current set of reached states then two different sets of successor states are computed —τ using the full transition relation T and τ_a using an ample transition relation $T_a \subseteq T$. Since $T_a \subseteq T$, we have that $\tau_a \subseteq \tau$. A heuristic BDD minimization function available in the BDD package in VIS is then used to find a set of states σ' with small BDD representation such that $\sigma \cup \tau_a \subseteq \sigma' \subseteq \sigma \cup \tau$. The current reached set is updated to be σ'. Clearly, this approach also produces a partial-order reduction of the transition system. In both examples discussed in Section 3, the heuristic minimization always returned the smaller reached set, that is, the one obtained from the ample transition relation. We believe that the reason for this is the exponential reduction in the number of states obtained using ample sets. However, if a system has asynchronous components operating with limited independence and there is limited partial-order reduction, this method might be better than just using the ample transition relation for the reachability computation.

Acknowledgments

Gerard Holzmann provided us with information on SPIN. Ken McMillan and Doron Peled contributed through discussions. The VIS group at UC Berkeley and Rajeev Ranjan in particular helped with the experiments.

References

[BCL91] J.R. Burch, E.M. Clarke, and D.E. Long. Symbolic model checking with partitioned transition relations. In *Proc. of the 28th Design Automation Conference*, pages 403–407, 1991.

[BCM+92] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.

[BHSV+96] R.K. Brayton, G.D. Hachtel, A.L. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S.-T. Cheng, S. Edwards, S. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R.K. Ranjan, S. Sarwary, T.R. Shiple, G.M. Swamy, and T. Villa. VIS: A system for verification and synthesis. In R. Alur and T.A. Henzinger, editors, *CAV 96: Computer Aided Verification*, Lecture Notes in Computer Science 1102, pages 428–432. Springer-Verlag, 1996.

[CP96] C.-T. Chou and D.A. Peled. Formal verification of a partial-order reduction technique for model checking. In *Proc. of the 2nd International Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1055 of *Lecture Notes in Computer Science*, pages 241–257. Springer-Verlag, 1996.

[DKR82] D. Dolev, M. Klawe, and M. Rodeh. An $o(n \log n)$ unidirectional distributed algorithm for extrema finding in a circle. *Journal of Algorithms*, 3:245–260, 1982.

[God96] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*, volume 1032 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[GW94] P. Godefroid and P. Wolper. A partial approach to model checking. *Information and Computation*, 110(2):305–326, 1994.

[HP94] G.J. Holzmann and D.A. Peled. An improvement in formal verification. In *Proc. of the 7th International Conference on Formal Description Techniques*, pages 197–211. Chapman & Hall, 1994.

[Maz89] A. Mazurkiewicz. Basic notions of trace theory. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Linear Time, Branching Time, and Partial Order in Logics and Models for Concurrency*, Lecture Notes in Computer Science 354, pages 285–363. Springer-Verlag, 1989.

[McM93] K.L. McMillan. *Symbolic Model Checking: An Approach to the State-Explosion Problem.* Kluwer Academic Publishers, 1993.

[Pel93] D. Peled. All from one, one for all: on model checking using representatives. In C. Courcoubetis, editor, *CAV 93: Computer Aided Verification*, Lecture Notes in Computer Science 697, pages 409–423. Springer-Verlag, 1993.

[RAP+95] R.K. Ranjan, A. Aziz, B. Plessier, C. Pixley, and R.K. Brayton. Efficient formal design verification: data structures + algorithms. In *Workshop Notes of the International Workshop on Logic Synthesis*, 1995.

[Val90] A. Valmari. A stubborn attack on state explosion. In R.P. Kurshan and E.M. Clarke, editors, *CAV 90: Computer Aided Verification*, Lecture Notes in Computer Science 531, pages 25–42. Springer-Verlag, 1990.

A Symbolic Computation of Ample Sets

Symbolic techniques capture transition sets and state sets of a labeled transition graph as boolean functions. The boolean functions are represented and manipulated as BDDs [BCM+92, McM93]. The functional representation of the transition relation is denoted by T, where $T(s, \alpha, t)$ is true iff $s \xrightarrow{\alpha} t$. Note that the BDD for T is built with variables that encode the present state, the next state, and the actions of G (these variables should not be confused with the variables of the labeled transition graph). In BFS, we denote the set of reached states by σ. First, we initialize σ to be the set of initial states. Second, the image σ_{new} of σ under T is computed: $\sigma_{new}(s) = \exists t.\exists\alpha.(\sigma(t) \;\wedge\; T(t, \alpha, s))$. The set of states that can be reached by the system in at most one step from σ is represented by $\sigma \vee \sigma_{new}$. The image computations are iterated until a fixpoint is reached. This fixpoint then represents the set of all reachable states. Figure 12 shows the modified BFS algorithm from Figure 1 (with partial-order reduction) rewritten with BDD operations. To implement this algorithm we need to produce an *ample transition relation* T^A at each reachability step.

A *persistent transition relation* T^P is a BDD encoding of a persistent function Δ such that $T^P(s, \alpha, t)$ iff $\alpha \in \Delta(s)$ and $s \xrightarrow{\alpha} t$. An ample transition relation T^A with respect to the history σ

σ_{new} := BDD representing $\hat{S}$
repeat
  /* σ is the current set of reached states */
  $\sigma := \sigma_{new}$
  $\mathcal{F}$:= BDD representing the frontier of σ
  T^A := BDD representing the ample transition relation with respect to the history σ
  σ_{next} := image of $\mathcal{F}$ under T^A
  $\sigma_{new} := \sigma \vee \sigma_{next}$
until $\sigma = \sigma_{new}$

Figure 12: BDD-based modified breadth-first search algorithm

can be computed from T^P as

$$T^A(s, \alpha, t) \;=\; T^P(s, \alpha, t) \vee (\neg\exists\alpha'.\exists t'.(T^P(s, \alpha', t') \wedge \neg\sigma(t')) \wedge T(s, \alpha, t)).$$

It remains to be seen how we compute the persistent transition relation T^P. We now show that a heuristic for computing persistent sets in [God96] can be implemented symbolically to yield a persistent transition relation. We assume a model of the transition system annotated with processes and give a symbolic procedure for computing persistent sets in this model.

A *process structure* is a structured description of a labeled transition graph. Formally, it is a 4-tuple $\mathcal{P} = \langle \mathbb{P}, G, own, exec \rangle$ with the following components:

- A finite set $\mathbb{P} = \{p_1, p_2, \ldots, p_n\}$ of processes.
- A labeled transition graph $G = \langle V, S, \hat{S}, \Gamma, \rightarrow \rangle$.
- A function *own* from V to $\mathbb{P}$. We say that a variable v is *owned* by process $own(v)$.
- A function *exec* from Γ to $\mathbb{P}$, with the following restriction: for all states s and t, all actions α, and all variables v, if $s \xrightarrow{\alpha} t$ and v changes value from s to t, then $own(v) = exec(\alpha)$. We say that action α is *executed* by process $exec(\alpha)$.

Observe that in this model, variables are read-shared, but write-exclusive. We write $act(p_i)$ to denote the set $\{\alpha \mid exec(\alpha) = p_i\}$ of actions that are executed by process p_i. Since each action is executed by exactly one process, *act* induces a partition on the set Γ of actions. We differentiate between *global state* (valuations for all variables) and *local state* of a process (valuations for the variables owned by the process). A global state s is an n-tuple $(s_1, s_2, \ldots, s_n)$ of local states s_i, one for each process $p_i \in \mathbb{P}$. We say that s_i is the local state of process p_i in s. An action $\alpha \in act(p_i)$ is *enabled in the local state* s_i of process p_i if there exists a global state s containing s_i such that α is enabled in s.

The algorithm for computing persistent sets is sketched in Figure 13. Given a state s, the algorithm finds a set of processes $P(s) \subseteq \mathbb{P}$ such that the set of actions executed by the processes

```
/* Let s = (s_1, s_2, ..., s_n) be the given global state */
P(s) := {p_k}, where p_k is some process enabled in s
repeat
    for every process p_i ∈ P(s) and every action α ∈ act(p_i) enabled in s_i do
        for every process p_j ∉ P(s) do
            if there exists an action β ∈ act(p_i) such that β is dependent with α then
                P(s) := P(s) ∪ {p_j}
            fi
        end
    end
until no more processes can be added to P(s)
```

Figure 13: Algorithm for computing a persistent set $P(s)$ at state s

in $P(s)$ is a persistent set at s. First, we initialize $P(s)$ with an arbitrary process p_k that executes an action enabled in s. Then we iteratively add more processes to $P(s)$, as follows, until no more processes can be added: for all processes $p_i \in P(s)$ and actions $\alpha \in act(p_i)$ that are enabled in the local state s_i of process p_i, we add process p_j to $P(s)$ if $act(p_j)$ executes an action β that is dependent with α. We call the resulting set $P(s)$ a *persistent set of processes* at s. The algorithm described above checks only a sufficient condition for persistence; it is possible to formulate more complex schemes that yield smaller persistent sets [God96].

Our aim is to compute a boolean function I such that $I(s,p)$ is true iff $p \in P(s)$ for the global state s. In effect, I encodes persistent sets of processes for all global states. For computing I, we make use of the following pre-computed relations:

- $enabledG(s,\alpha)$ is true iff action α is enabled in the global state s.
- $actionof(\alpha,p)$ is true iff $exec(\alpha) = p$.
- $enabledL(s,\alpha,p)$ is true iff $exec(\alpha) = p$ and α is enabled in the local state of p in s:

$$\begin{aligned} enabledL((s_1,s_2,s_3,...,s_n),\alpha,p) \quad = \quad & \bigvee_{1\le i\le n}((p=p_i) \Rightarrow \\ & \exists s'_1 \ldots s'_{i-1} s'_{i+1} \ldots s'_n . \\ & enabledG((s'_1,s'_2,\ldots,s'_{i-1},s_i,s'_{i+1},\ldots,s'_n),\alpha)). \end{aligned}$$

- $dependent(\alpha,\beta)$ is true iff actions α and β are dependent. The dependence relation can be overapproximated by sufficient syntactic checks (e.g., disjointness of support variables) [God96].

For all i, $1 \le i \le n$, we compute $J_i(s,p)$ as follows:

$$J_i(s,p) \;=\; (p = p_i) \wedge \exists \alpha.(actionof(\alpha,p_i) \wedge enabledG(s,\alpha)).$$

The relation $J_i(s,p)$ is true iff $p = p_i$ and $act(p_i)$ contains at least action enabled in s. We then perform a fixpoint computation to compute the relation $I(s,p)$ as follows:

$$\begin{aligned} I_0(s,p) \quad = \quad & J_1(s,p) \vee \\ & (\neg\exists p. J_1(s,p) \wedge J_2(s,p)) \vee \\ & (\neg\exists p. (J_1(s,p) \vee J_2(s,p)) \wedge J_3(s,p)) \ \vee \ldots \\ & (\neg\exists p. (J_1(s,p) \vee J_2(s,p) \vee \ldots \vee J_{n-1}(s,p)) \wedge J_n(s,p)), \end{aligned}$$

$$\begin{aligned} I_{k+1}(s,p) \quad = \quad & I_k(s,p) \vee \\ & \exists p'.\exists\alpha.\exists\beta.(I_k(s,p') \wedge \mathit{enabledL}(s,\alpha,p') \wedge \ \mathit{actionof}(\alpha',p) \wedge \mathit{dependent}(\alpha,\beta)). \end{aligned}$$

The relation $I_0(s,p)$ is true iff p is the process of least index that executes an action enabled in state s. The computation of I_{k+1} from I_k mimics the repeat-until loop of the algorithm in Figure 13. Then, the desired relation I is I_m for the smallest m such that $I_{m+1} = I_m$. Now the persistent transition relation T^P can be computed from I as follows:

$$T^P(s,\alpha,t) \ = \ \exists p.(I(s,p) \wedge \mathit{actionof}(\alpha,p)) \wedge T(s,\alpha,t).$$