Efficient Detection of Global Properties in Distributed Systems Using Partial-Order Methods*

Scott D. Stoller, Leena Unnikrishnan, and Yanhong A. Liu

Computer Science Dept., Indiana University, Bloomington, IN 47405-7104 USA

Abstract. A new approach is presented for detecting whether a particular computation of an asynchronous distributed system satisfies **Poss** Φ (read "possibly Φ"), meaning the system could have passed through a global state satisfying predicate Φ, or **Def** Φ (read "definitely Φ"), meaning the system definitely passed through a global state satisfying Φ. Detection can be done easily by straightforward state-space search; this is essentially what Cooper and Marzullo proposed. We show that the persistent-set technique, a well-known partial-order method for optimizing state-space search, provides efficient detection. This approach achieves the same worst-case asymptotic time complexity as two special-purpose detection algorithms of Garg and Waldecker that detect **Poss** Φ and **Def** Φ for a restricted but important class of predicates. For **Poss** Φ, our approach applies to arbitrary predicates and thus is more general than Garg and Waldecker's algorithm. We apply our algorithm for **Poss** Φ to two examples, achieving a speedup of over 700 in one example and over 70 in the other, compared to unoptimized state-space search.

1 Introduction

Detecting global properties (*i.e.*, predicates on global states) in distributed systems is useful for monitoring and debugging. For example, when testing a distributed mutual exclusion algorithm, it is useful to monitor the system to detect concurrent accesses to the critical sections. A system that performs leader election may be monitored to ensure that processes agree on the current leader. A system that dynamically partitions and re-partitions a large dataset among a set of processors may be monitored to ensure that each portion of the dataset is assigned to exactly one processor.

An *asynchronous* distributed system is characterized by lack of synchronized clocks and lack of bounds on processor speed and network latency. In such a system, no process can determine in general the order in which events on different processors actually occurred. Therefore, no process can determine in general the

* The authors gratefully acknowledge the support of NSF under Grants CCR-9876058 and CCR-9711253 and the support of ONR under Grants N00014-99-1-0358 and N00014-99-1-0132. Email: {stoller,lunnikri,liu}@cs.indiana.edu Web: http://www.cs.indiana.edu/~{stoller,lunnikri,liu}/

E.A. Emerson and A.P. Sistla (Eds.): CAV 2000, LNCS 1855, pp. 264–279, 2000.

sequence of global states through which the system passed. This leads to an obvious difficulty for detecting whether a global property held.

Cooper and Marzullo's solution to this difficulty involves two modalities, which we denote by **Poss** (read "possibly") and **Def** (read "definitely") [CM91]. These modalities are based on logical time as embodied in the *happened-before* relation, a partial order that reflects causal dependencies [Lam78]. A history of an asynchronous distributed system can be approximated by a *computation*, which comprises the local computation of each process together with the happened-before relation. Happened-before is useful for detection algorithms because, using vector clocks [Fid88,Mat89,SW89], it can be determined by processes in the system.

Happened-before is not a total order, so it does not uniquely determine the history. But it does restrict the possibilities. Histories *consistent* with a computation c are those sequences of the events in c that correspond to total orders containing the happened-before relation. A *consistent global state* (CGS) of a computation c is a global state that appears in some history consistent with c. A computation c satisfies $\mathbf{Poss}\,\Phi$ iff, in *some* history consistent with c, the system passes through a global state satisfying Φ. A computation c satisfies $\mathbf{Def}\,\Phi$ iff, in *all* histories consistent with c, the system passes through a global state satisfying Φ.

Cooper and Marzullo give centralized algorithms for detecting $\mathbf{Poss}\,\Phi$ and $\mathbf{Def}\,\Phi$ [CM91]. A stub at each process reports the local states of that process to a central monitor. The monitor incrementally constructs a lattice whose elements correspond to CGSs of the computation. $\mathbf{Poss}\,\Phi$ and $\mathbf{Def}\,\Phi$ are evaluated by straightforward traversals of the lattice. In a system of N processes, the worst-case number of CGSs, which can occur in computations containing little communication, is $\Theta(S^N)$, where S is the maximum number of steps taken by a single process. Any detection algorithm that enumerates all CGSs—like the algorithms in [CM91,MN91,JMN95,AV94]—has time complexity that is at least linear in the number of CGSs. This time complexity can be prohibitive. This motivated the development of efficient algorithms for detecting restricted classes of predicates [TG93,GW94,GW96,CG98]. The algorithms of Garg and Waldecker are classic examples of this approach. A predicate is *n-local* if it depends on the local states of at most n processes. In [GW94] and [GW96], Garg and Waldecker give efficient algorithms that detect $\mathbf{Poss}\,\Phi$ and $\mathbf{Def}\,\Phi$, respectively, for predicates Φ that are conjunctions of 1-local predicates. Those two algorithms are presented as two independent works, with little relationship to each other or to existing techniques.

This paper shows that efficient detection of global predicates can be done using a well-known partial-order method. *Partial-order methods* are optimized state-space search algorithms that try to avoid exploring multiple interleavings of independent transitions [PPH97]. This approach achieves the same worst-case asymptotic time complexity as the two aforementioned algorithms of Garg and Waldecker, assuming weak vector clocks [MN91], which are updated only by events that can change the truth value of Φ and by receive events by which

a process first learns of some event that can change the truth value of Φ, are used with our algorithm. Specifically, we show that persistent-set selective search [God96] can be used to detect $\mathbf{Poss}\,\Phi$ or $\mathbf{Def}\,\Phi$ for conjunctions of 1-local predicates with time complexity $O(N^2S)$. In some non-worst cases, Garg and Waldecker's algorithms may be faster than ours by up to a factor of N, because their algorithms also incorporate an idea, not captured by partial-order methods, by which the algorithm ignores local states of a process that do not satisfy the 1-local predicate associated with that process. For details, see [SUL99].

Our method for detecting $\mathbf{Def}\,\Phi$ handles only conjunctions of 1-local predicates. Our method for detecting $\mathbf{Poss}\,\Phi$ handles arbitrary predicates and thus is more general than Garg and Waldecker's algorithm. Furthermore, our method is asymptotically faster than Cooper and Marzullo's algorithm for some classes of systems to which Garg and Waldecker's algorithm does not apply. For some other classes of systems, although our method has the same asymptotic worst-case complexity as Cooper and Marzullo's algorithm, we expect our method to be significantly faster in practice; this is typical of general experience with partial-order methods. Our algorithm for detecting $\mathbf{Poss}\,\Phi$ can be further optimized to sometimes explore sequences of transitions in a single step. This can provide significant speedup, even reducing the asymptotic time and space complexities for certain classes of systems.

We give simple specialized algorithms $\mathrm{PS}_{\mathbf{Poss}}$ and $\mathrm{PS}_{\mathbf{Def}}$ for computing persistent sets for detection of $\mathbf{Poss}\,\Phi$ and $\mathbf{Def}\,\Phi$, respectively. These algorithms exploit the structure of the problem in order to efficiently compute small persistent sets. One could instead use a general-purpose algorithm for computing persistent sets, such as the conditional stubborn set algorithm (CSSA) [God96, Section 4.7], which is based on Valmari's work on stubborn sets [Val97]. When CSSA is used for detecting $\mathbf{Poss}\,\Phi$, it is either ineffective (*i.e.*, it returns the set of all enabled transitions) or slower than $\mathrm{PS}_{\mathbf{Poss}}$ by a factor of S (and possibly by some factors of N) in the worst case, depending on how it is applied. The cheaper algorithms for computing persistent sets in [God96] are ineffective for detecting $\mathbf{Poss}\,\Phi$. When CSSA (or any of the other algorithms in [God96]) is used for detecting $\mathbf{Def}\,\Phi$, it is ineffective. Detailed justifications of these claims appear in [SUL99].

For simplicity, we present algorithms for *off-line* property detection, in which the detection algorithm is run after the distributed computation has terminated. Our approach can also be applied to *on-line* property detection, in which a monitor runs concurrently with the system being monitored.

Property detection is a special case of model-checking of temporal logics interpreted over partially-ordered sets of global configurations, as described in [AMP98,Wal98]. Those papers do not discuss in detail the use of partial-order methods to avoid exploring all global states and do not characterize classes of global predicates for which partial-order methods reduce the worst-case asymptotic time complexity. Alur *et al.* give a decision procedure for the logic $\mathrm{ISTL}^{\diamond}$. $\mathbf{Poss}\,\Phi$ is expressible in $\mathrm{ISTL}^{\diamond}$ as $\exists\diamond\Phi$. $\mathbf{Def}\,\Phi$ is expressible in ISTL as $\neg\exists\Box\neg\Phi$ but appears not to be directly expressible in $\mathrm{ISTL}^{\diamond}$.

An avenue for future work is to try to extend this approach to efficient analysis of message sequence charts [MPS98,AY99].

2 Background on Property Detection

A local state of a process is a mapping from identifiers to values. Thus, $s(v)$ denotes the value of variable v in local state s. A *history* of a single process is represented as a sequence of that process's states. Let $[m..n]$ denote the set of integers from m to n, inclusive. We use integers $[1..N]$ as process names.

In the distributed computing literature, the most common representation of a *computation* c of an asynchronous distributed system is a collection of histories $c[1], \ldots, c[N]$, one for each constituent process, together with a *happened-before* relation $\rightarrow$ on local states [GW94]. For a sequence h, let $h[k]$ denote the k^{th} element of h (*i.e.*, we use 1-based indexing), and let $|h|$ denote the length of h. Intuitively, a local state s_1 happened-before a local state s_2 if s_1 finished before s_2 started. Formally, $\rightarrow$ is the smallest transitive relation on the local states in c such that

1. $\forall i \in [1..N], k \in [1..|c[i]| - 1] \ : \ c[i][k] \rightarrow c[i][k+1])$.
2. For all local states s_1 and s_2 in c, if the event immediately following s_1 is the sending of a message and the event immediately preceding s_2 is the reception of that message, then $s_1 \rightarrow s_2$.

We always use S to denote the maximum number of local states per process, *i.e.*, $\max(|c[1]|, \ldots, |c[N]|)$.

Each process has a distinguished variable vt such that for each local state s, $s(vt)$ is a *vector timestamp* [Mat89], *i.e.*, an array of N natural numbers such that $s(vt)[i]$ is the number of local states of process i that happened-before s. Vector timestamps capture the happened-before relation. Specifically, for all local states s_1 and s_2, $s_1 \rightarrow s_2$ iff $(\forall i \in [1..N] : s_1(vt)[i] \leq s_2(vt)[i])$. Two local states s_1 and s_2 of a computation are *concurrent*, denoted $s_1 \parallel s_2$, iff neither happened-before the other: $s_1 \parallel s_2 \ = \ s_1 \not\rightarrow s_2 \ \wedge \ s_2 \not\rightarrow s_1$.

A *global state* s of a computation c is an array of N local states such that, for each process i, $s[i]$ appears in $c[i]$. A global state is *consistent* iff its constituent local states are pairwise concurrent. Intuitively, consistency means that the system could have passed through that global state during the computation.

Concurrency of two local states can be tested in constant time using vector timestamps by exploiting the following theorem [FR94]: for a local state s_1 of process i_1 and a local state s_2 of process i_2, $s_1 \parallel s_2$ iff $s_2(vt)[i_2] \geq s_1(vt)[i_2] \ \wedge \ s_1(vt)[i_1] \geq s_2(vt)[i_1]$. Thus, a global state s is consistent iff $(\forall i, j \in [1..N] : s[i](vt)[i] \geq s[j](vt)[i])$.

A computation c satisfies $\mathbf{Poss}\,\Phi$, denoted $c \models \mathbf{Poss}\,\Phi$, iff there exists a CGS of c that satisfies Φ.

Introduce a partial order $\prec_G$ on global states: $s_1 \preceq_G s_2 \ = \ (\forall i \in [1..N] : s_1[i] = s_2[i] \vee s_1[i] \rightarrow s_2[i])$. A *history consistent with* a computation c is a finite or infinite sequence σ of consistent global states of c such that, with respect to

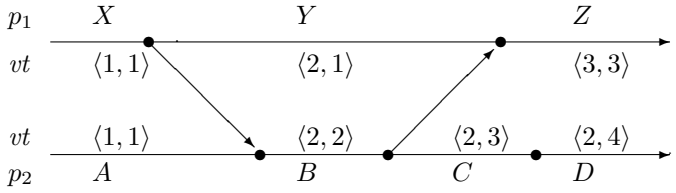

Fig. 1. Computation c_0.

$\preceq_G$: (*i*) $\sigma[1]$ is minimal; (*ii*) for all $k \in [1..|\sigma| - 1]$, $\sigma[k+1]$ is an immediate successor[1] of $\sigma[k]$; and (*iii*) if σ is finite, then $\sigma[|\sigma|]$ is maximal.

A computation c satisfies $\mathbf{Def}\,\Phi$, denoted $c \models \mathbf{Def}\,\Phi$, iff every history consistent with c contains a global state satisfying Φ.

Example. Consider the computation c_0 shown in Figure 1. Horizontal lines correspond to processes; diagonal lines, to messages. Dots represent events. Each process has a variable vt containing the vector time. Variable p_i contains the rest of process i's local state. Let s_{k_1,k_2} denote the global state comprising the k_1'th local state of process 1 and the k_2'th local state of process 2. The CGSs of c_0 are $\{s_{1,1}, s_{2,1}, s_{2,2}, s_{2,3}, s_{3,3}, s_{2,4}, s_{3,4}\}$. Some properties of this computation are: $c_0 \models \mathbf{Poss}\,(p_1 = Y \wedge p_2 = D)$, $c_0 \models \mathbf{Def}\,(p_1 = Y \wedge p_2 = B)$, $c_0 \not\models \mathbf{Def}\,(p_1 = Y \wedge p_2 = D)$, and $c_0 \not\models \mathbf{Poss}\,(p_1 = X \wedge p_2 = B)$.

3 Background on Partial-Order Methods

The material in this section is paraphrased from [God96]. Beware! The system model in this section differs from the model of distributed computations in the previous section. For example, "state" has different meanings in the two models. Sections 4 and 5 give mappings from the former model to the latter.

A concurrent system is a collection of finite-state automata that interact via shared variables (more generally, shared objects). More formally, a *concurrent system* is a tuple $\langle \mathcal{P}, \mathcal{O}, \mathcal{T}, s_{init} \rangle$, where

- $\mathcal{P}$ is a set $\{P_1, \ldots, P_N\}$ of processes. A *process* is a finite set of control points.
- $\mathcal{O}$ is a set of shared variables.
- $\mathcal{T}$ is a set of transitions. A *transition* is a tuple $\langle L_1, G, C, L_2 \rangle$, where: L_1 is a set of control points, at most one from each process; L_2 is a set of control points of the same processes as L_1, and with at most one control point from each process; G is a guard, *i.e.*, a boolean-valued expression over the shared variables; and C is a command, *i.e.*, a sequence of operations that update the shared variables.
- s_{init} is the initial state of the system.

[1] For a reflexive or irreflexive partial order $\langle S, \prec \rangle$ and elements $x \in S$ and $y \in S$, y is an *immediate successor* of x iff $x \neq y \wedge x \prec y \wedge \neg(\exists\, z \in S \setminus \{x, y\} : x \prec z \wedge z \prec y)$.

A (global) state is a tuple $\langle L, V\rangle$, where L is a collection of control points, one from each process, and V is a collection of values, one for each shared variable. For a state s and a shared variable v, we abuse notation and write $s(v)$ to denote the value of v in s (the same notation is used in Section 2 but with a different definition of "state"). Similarly, for a state s and a predicate ϕ, we write $s(\phi)$ to denote the value of ϕ in s. A transition $\langle L_1, G, C, L_2\rangle$ is *enabled* in state $\langle L, V\rangle$ if $L_1 \subseteq L$ and G evaluates to true using the values in V. Let $enabled(s)$ denote the set of transitions enabled in s. If a transition $\langle L_1, G, C, L_2\rangle$ is enabled in state $s = \langle L, V\rangle$, then it can be executed in s, leading to the state $\langle (L \setminus L_1) \cup L_2, C(V)\rangle$, where $C(V)$ represents the new values obtained by using the operations in C to update the values in V. We write $s \xrightarrow{t} s'$ to indicate that transition t is enabled in state s and executing t in s leads to state s'.

An *execution* of a concurrent system is a finite or infinite sequence $s_1 \xrightarrow{t_1} s_2 \xrightarrow{t_2} s_3 \cdots$ such that $s_1 = s_{init}$ and for all i, $s_i \xrightarrow{t_i} s_{i+1}$. A state is *reachable* (in a system) if it appears in some execution (of that system).

Suppose we wish to find all the "deadlocks" of a system. Following Godefroid (but deviating from standard usage), a *deadlock* is a state in which no transitions are enabled. Clearly, all reachable deadlocks can be identified by exploring all reachable states. This involves explicitly considering all possible execution orderings of transitions, even if some transitions are "independent" (*i.e.*, executing them in any order leads to the same state; formal definition appears in Appendix). Exploring one interleaving of independent transitions is sufficient for finding deadlocks. This does cause fewer intermediate states (*i.e.*, states in which some but not all of the independent transitions have been executed) to be explored, but it does not affect reachability of deadlocks, because the intermediate states cannot be deadlocks, because some of the independent transitions are enabled in those states. Partial-order methods attempt to eliminate exploration of multiple interleavings of independent transitions, thereby saving time and space.

A set T of transitions enabled in a state s is persistent in s if, for every sequence of transitions starting from s and not containing any transitions in T, all transitions in that sequence are independent with all transitions in T. Formally, a set $T \subseteq enabled(s)$ is *persistent* in s iff, for all nonempty sequences of transitions $s_1 \xrightarrow{t_1} s_2 \xrightarrow{t_2} s_3 \cdots \xrightarrow{t_{n-1}} s_n \xrightarrow{t_n} s_{n+1}$, if $s_1 = s$ and $(\forall i \in [1..n] : t_i \notin T)$, then t_n is independent in s_n with all transitions in T. As shown in [God96], in order to find all reachable deadlocks, it suffices to explore from each state s a set of transitions that is persistent in s. State-space search algorithms that do this are called *persistent-set selective search* (PSSS). Note that $enabled(s)$ is trivially persistent in s. To save time and space, small persistent sets should be used.

4 Detecting Poss Φ

Given a computation c and a predicate Φ, we construct a concurrent system whose executions correspond to histories consistent with c, express $c \models \mathbf{Poss}\,\Phi$

as a question about reachable deadlocks of that system, and use PSSS to answer that question. The system has one transition for each pair of consecutive local states in c, plus a transition t_0 whose guard is Φ. t_0 disables all transitions, so it always leads to a deadlock.

Each process has a distinct control point corresponding to each of its local states. The control point corresponding to the k'th local state of process i is denoted $\ell_{i,k}$. Thus, for $i \in [1..N]$, process i is $P_i = \bigcup_{k=1..|c[i]|}\{\ell_{i,k}\}$. We introduce a new process, called process 0, that monitors Φ. Process 0 has a single transition t_0, which changes the control point of process 0 from $\ell_{0,nd}$ (mnemonic for "not detected") to $\ell_{0,d}$ ("detected"). Thus, process 0 is $P_0 = \{\ell_{0,nd}, \ell_{0,d}\}$. The set of processes is $\mathcal{P} = \bigcup_{i=0..N}\{P_i\}$. Initially, process 0 is at control point $\ell_{0,nd}$, and for $i > 0$, process i is at control point $\ell_{i,1}$.

The local state of process i is stored in a shared variable p_i. The initial value of p_i is $c[i][1]$. For convenience, the index of the current local state of process i is stored in a shared variable τ_i. The initial value of τ_i is 1, and τ_i is incremented by each transition of process i. Whenever process i is at control point $\ell_{i,k}$, τ_i equals k. The set of shared variables is $\mathcal{O} = \bigcup_{i=1..N}\{p_i, \tau_i\}$.

Transition $t_{i,k}$ takes process i from its k'th local state to its $(k+1)$'th local state. $t_{i,k}$ is enabled when process i is at control point $\ell_{i,k}$, process 0 is at control point $\ell_{0,nd}$, τ_i equals k, and the $(k+1)$'th local state of process i is concurrent with the current local states of the other processes. The set of transitions is $\mathcal{T} = \{t_0\} \cup \bigcup_{i=1..N,k=1..|c[i]|-1}\{t_{i,k}\}$, where $t_0 = \langle\{l_{0,nd}\}, \Phi(p_1, \ldots, p_N), \text{skip}, \{l_{0,d}\}\rangle$ and

$$\begin{aligned} t_{i,k} = \langle &\{\ell_{i,k}, \ell_{0,nd}\}, \\ &\tau_i = k \ \wedge\ (\forall j \in [1..N] \setminus \{i\} : c[j][\tau_j](vt)[j] \geq c[i][k+1](vt)[j]), \\ &p_i := c[i][k+1];\ \tau_i := k+1, \\ &\{\ell_{i,k+1}, \ell_{0,nd}\}\rangle \end{aligned} \tag{1}$$

The guard can be simplified by noting that $c[i][\tau_i](vt)[i]$ always equals τ_i. It is easy to show that $\ell_{0,d}$ is reachable iff a state satisfying Φ is reachable, and that all states containing $\ell_{0,d}$ are deadlocks. Thus, $c \models \mathbf{Poss}\,\Phi$ iff a deadlock containing $\ell_{0,d}$ is reachable.

Example. The transitions of the concurrent system corresponding to c_0 of Figure 1 are t_0 and

$$\begin{aligned} t_{1,1} &= \langle\{\ell_{1,1}, \ell_{0,nd}\}, \tau_1 = 1 \ \wedge\ \tau_2 \geq 1, p_1 := Y;\ \tau_1 := 2, \{\ell_{1,2}, \ell_{0,nd}\}\rangle \\ t_{1,2} &= \langle\{\ell_{1,2}, \ell_{0,nd}\}, \tau_1 = 2 \ \wedge\ \tau_2 \geq 3, p_1 := Z;\ \tau_1 := 3, \{\ell_{1,3}, \ell_{0,nd}\}\rangle \\ t_{2,1} &= \langle\{\ell_{2,1}, \ell_{0,nd}\}, \tau_2 = 1 \ \wedge\ \tau_1 \geq 2, p_2 := B;\ \tau_2 := 2, \{\ell_{2,2}, \ell_{0,nd}\}\rangle \\ t_{2,2} &= \langle\{\ell_{2,2}, \ell_{0,nd}\}, \tau_2 = 2 \ \wedge\ \tau_1 \geq 2, p_2 := C;\ \tau_2 := 3, \{\ell_{2,3}, \ell_{0,nd}\}\rangle \\ t_{2,3} &= \langle\{\ell_{2,3}, \ell_{0,nd}\}, \tau_2 = 3 \ \wedge\ \tau_1 \geq 2, p_2 := D;\ \tau_2 := 4, \{\ell_{2,4}, \ell_{0,nd}\}\rangle \end{aligned} \tag{2}$$

An alternative is to construct a transition system similar to the one above but in which both occurrences of $\ell_{0,nd}$ in $t_{i,k}$ are deleted. As before, $c \models \mathbf{Poss}\,\Phi$ iff $\ell_{0,d}$ is reachable (or, equivalently, t_0 is reachable), but now, states containing $\ell_{0,d}$ are not necessarily deadlocks. PSSS can be used to determine reachability

of control points (or transitions), provided the dependency relation is weakly uniform [God96, Section 6.3]. Showing that the dependency relation is weakly uniform requires more effort than including $\ell_{0,nd}$ in $t_{i,k}$ and has no benefit: we obtain essentially the same detection algorithm either way.

We give a simple algorithm $\mathrm{PS}_{\mathbf{Poss}}$ that efficiently computes a small persistent set in a state s by exploiting the structure of Φ. Without loss of generality, we write Φ as a conjunction: $\Phi = \bigwedge_{i=1..n} \phi_i$, with $n \geq 1$. The *support* of a formula ϕ, denoted $supp(\phi)$, is the set of processes on whose local states ϕ depends. Suppose Φ is true in s. Then $\mathrm{PS}_{\mathbf{Poss}}(s)$ returns $enabled(s)$. When such a state is reached, there is no need to try to find a small persistent set, because we can immediately halt the search and return "$c \models \mathbf{Poss}\,\Phi$". We return "$c \not\models \mathbf{Poss}\,\Phi$" if the selective search terminates without encountering a state satisfying Φ; by construction, this is equivalent to unreachability of deadlocks containing $\ell_{0,d}$. Suppose Φ is false in s. The handling of this case is based on the following theorem, a proof of which appears in the Appendix.

Theorem 1. *Suppose Φ is false in s. Let T be a subset of $enabled(s)$ such that, for all sequences of transitions starting from s and staying outside T, Φ remains false; more precisely, for all sequences of transitions $s_1 \xrightarrow{t_1} s_2 \xrightarrow{t_2} \cdots s_n \xrightarrow{t_n} s_{n+1}$, if $s_1 = s$ and $(\forall i \in [1..n] : t_i \notin T)$, then Φ is false in s_{n+1}. Then T is persistent in s.*

To construct such a set T, choose some conjunct ϕ of Φ that is false in s. Clearly, Φ cannot become true until ϕ does, and ϕ cannot become true until the next transition of some process in $supp(\phi)$ is executed. Note that the next transition of process i must be $t_{i,s(\tau_i)}$. Thus, for each process i in $supp(\phi)$, if process i is not in its final state (*i.e.*, $s(\tau_i) < |c[i]|$), and if the next transition $t_{i,s(\tau_i)}$ of process i is enabled, then add $t_{i,s(\tau_i)}$ to T, otherwise find some enabled transition t that must execute before $t_{i,s(\tau_i)}$, and add t to T. To find such a t, we introduce the *wait-for graph* $WF(s)$, which has nodes $[1..N]$, and has an edge from i to j if the next transition of process j must execute before the next transition of process i, *i.e.*, if $s(\tau_j) < c[i][s(\tau_i)+1](vt)[j]$ (we call this a wait-for graph because of its similarity to the wait-for graphs used for deadlock detection [SG98, Section 7.6.1]). Such a transition t can be found by starting at node i in $WF(s)$, following any path until a node j with no outedges is reached, and taking t to be $t_{j,s(\tau_j)}$. The case in which $t_{i,s(\tau_i)}$ is enabled is a special case of this construction, corresponding to a path of length zero, which implies $i = j$. Pseudo-code appears in Figure 2.

The wait-for graph can be incrementally maintained in $O(N)$ time per transition. Let $d = \max(|supp(\phi_1)|, \ldots, |supp(\phi_n)|)$. $\mathrm{PS}_{\mathbf{Poss}}(s)$ returns a set of size at most d if Φ is false in s. Computing $\mathrm{PS}_{\mathbf{Poss}}(s)$ takes $O(Nd)$ time, because the algorithm follows at most d paths of length at most N in the wait-for graph. Thus, the overall time complexity of the search is $O(NdN_e)$, where N_e is the number of states explored by the algorithm.

Suppose Φ is a conjunction of 1-local predicates. Then $\mathrm{PS}_{\mathbf{Poss}}$ returns sets of size at most 1, except when Φ is true in s, in which case the search is halted immediately, as described above. Thus, at most one transition is explored from

```
if (Φ holds in s) ∨ enabled(s) = ∅ then return enabled(s)
else choose some conjunct φ of Φ such that φ is false in s;
    T := ∅;
    for all i in supp(φ) such that s(τ_i) < |c[i]|
        start at node i in WF(s);
        follow any path until a node j with no outedges is reached;
        insert t_{j,s(τ_j)} in T;
    return T
```

Fig. 2. Algorithm $\mathrm{PS}_{\mathbf{Poss}}(s)$.

each state. Also, the system has a unique initial state. Thus, the algorithm explores one linear sequence of transitions. Each transition in $\mathcal{T}$ appears at most once in that sequence, because t_0 disables all transitions, and each transition $t_{i,k}$ permanently disables itself. $|\mathcal{T}|$ is $O(NS)$, so N_e is also $O(NS)$, so the overall time complexity is $O(N^2S)$.

Example. In evaluation of $c_0 \models \mathbf{Poss}\,(p_1 = X \wedge p_2 = B)$, $t_{1,1}$, $t_{2,1}$, and $t_{2,2}$ are executed. In the resulting state $s_{2,3}$, $\mathrm{PS}_{\mathbf{Poss}}(s_{2,3})$ may return $\{t_{1,2}\}$ or $\{t_{2,3}\}$, causing $s_{2,4}$ or $s_{3,3}$, respectively, to be never visited.

Example. Consider evaluation of $c \models \mathbf{Poss}\,\phi_1(p_1) \wedge \phi_2(p_2, p_3)$, for predicates ϕ_1 and ϕ_2 such that ϕ_1 is true in most states of process 1, and ϕ_2 is true in at most $O(S)$ consistent states of processes 2 and 3. PSSS does not explore transitions of process 1 in states where ϕ_2 is false and ϕ_1 is true, so its worst-case running time for such systems is $\Theta(S^2)$. Garg and Waldecker's algorithm [GW94] is inapplicable, because ϕ_2 is not 1-local. The worst-case running time of Cooper and Marzullo's algorithm [CM91] for such systems is $\Theta(S^3)$.

Exploring Sequences of Transitions. The following optimization can be used to reduce the number of explored states and transitions. In the **else** branch of $\mathrm{PS}_{\mathbf{Poss}}$, if the next transition of process i is not enabled, and if process i is not waiting for any other process in $supp(\phi)$, then insert in T a minimum-length sequence w of transitions that ends with a transition of process i. For details and an example, see [SUL99].

On-line Detection. For simplicity, the above presentation considers off-line property detection. Our approach can also be applied to on-line property detection. Local states arrive at the monitor one at a time. For each process, the local states of that process arrive in the order they occurred. However, there is no constraint on the relative arrival order of local states of different processes. For on-line detection of $\mathbf{Poss}\,\Phi$, detection must be announced as soon as local states comprising a CGS satisfying Φ have arrived. This is easily achieved by modifying the selective search algorithm to explore transitions as they become available: there is no need for the selective search to proceed in depth-first order, so the

stack can be replaced with a "to-do set", and in each iteration, any element of that set can be selected. This does not affect the time or space complexity of the algorithm.

5 Detecting Def Φ

As in Section 4, we construct a concurrent system whose executions correspond to histories consistent with c, express $c \models \mathbf{Def}\,\Phi$ as a question about reachable deadlocks of that system, and use PSSS to answer that question. The construction in Section 4 for **Poss** is similar to well-known constructions for reducing safety properties to deadlock detection [God96], but our construction for **Def** seems novel. The transitions are similar to those in (1), except the guard of each transition is augmented to check whether the transition would truthify Φ; if so, the transition is disabled. If $s_{init}(\Phi)$ holds, then $c \models \mathbf{Def}\,\Phi$, and no search is performed. Thus, in a search, Φ is false in all reachable states. If the final state (*i.e.*, the state satisfying $\bigwedge_{i=1..N} \tau_i = |c[i]|$) is reachable, then each sequence of transitions from s_{init} to the final state corresponds to a history consistent with c and in which Φ never holds, so $c \not\models \mathbf{Def}\,\Phi$.

The processes are the same as in Section 4, except with process 0 omitted. Thus, $\mathcal{P} = \bigcup_{i=1..N}\{P_i\}$, where $P_i = \bigcup_{k=1..|c[i]|}\{\ell_{i,k}\}$. The shared variables are the same as in Section 4. Thus, $\mathcal{O} = \bigcup_{i=1..N}\{p_i, \tau_i\}$. The initial state is the same as in Section 4, except with the control point for process 0 omitted. The transitions are $\mathcal{T} = \bigcup_{i=1..N,k=1..|c[i]|-1}\{t_{i,k}\}$, where

$$\begin{aligned} t_{i,k} = \langle &\{\ell_{i,k}\}, \\ &\tau_i = k \wedge (\forall j \in [1..N] \setminus \{i\} : c[j][\tau_j](vt)[j] \geq c[i][k+1](vt)[j]) \\ &\qquad \wedge \neg\Phi(p_1, \ldots, p_{i-1}, c[i][\tau_i + 1], p_{i+1}, \ldots, p[N]), \\ &p_i := c[i][k+1];\ \tau_i := k+1, \\ &\{\ell_{i,k+1}\}\rangle \end{aligned} \tag{3}$$

It is easy to show that $c \not\models \mathbf{Def}\,\Phi$ iff Φ is false in s_{init} and the final state is a reachable deadlock.

Example. The transitions of the concurrent system corresponding to computation c_0 of Figure 1 and the predicate $(p_1 = Y \wedge p_2 = D)$ are

$$\begin{aligned} t_{1,1} &= \langle\{\ell_{1,1}\},\ \tau_1 = 1 \ \wedge\ \tau_2 \geq 1 \ \wedge\ p_2 \neq D,\ p_1 := Y;\ \tau_1 := 2,\ \{\ell_{1,2}\}\rangle \\ t_{1,2} &= \langle\{\ell_{1,2}\},\ \tau_1 = 2 \ \wedge\ \tau_2 \geq 3,\ p_1 := Z;\ \tau_1 := 3,\ \{\ell_{1,3}\}\rangle \\ t_{2,1} &= \langle\{\ell_{2,1}\},\ \tau_2 = 1 \ \wedge\ \tau_1 \geq 2,\ p_2 := B;\ \tau_2 := 2,\ \{\ell_{2,2}\}\rangle \\ t_{2,2} &= \langle\{\ell_{2,2}\},\ \tau_2 = 2 \ \wedge\ \tau_1 \geq 2,\ p_2 := C;\ \tau_2 := 3,\ \{\ell_{2,3}\}\rangle \\ t_{2,3} &= \langle\{\ell_{2,3}\},\ \tau_2 = 3 \ \wedge\ \tau_1 \geq 2 \ \wedge\ p_1 \neq Y,\ p_2 := D;\ \tau_2 := 4,\ \{\ell_{2,4}\}\rangle \end{aligned} \tag{4}$$

Our algorithm $\text{PS}_{\mathbf{Def}}$ for computing persistent sets of such concurrent systems applies when Φ is a conjunction of 1-local predicates: $\Phi = \bigwedge_{i=1..N} \phi_i$, where $supp(\phi_i) = \{i\}$. Pseudo-code for $\text{PS}_{\mathbf{Def}}$ appears in Figure 3, where

$$stayfalse(i,k) = \neg c[i][k](\phi_i) \wedge \neg c[i][k+1](\phi_i). \tag{5}$$

if $enabled(s) = \emptyset$ **then return** $enabled(s)$
else choose some $i \in [1..N]$ such that $stayfalse(i, s(\tau_i))$;
start at node i in $WF(s)$;
follow any path until a node j with no outedges is reached;
return $\{t_{j,s(\tau_j)}\}$

Fig. 3. Algorithm $\mathrm{PS}_{\mathbf{Def}}(s)$.

Theorem 2. $\mathrm{PS}_{\mathbf{Def}}(s)$ *is well-defined and is persistent in* s.

Proof: To show that $\mathrm{PS}_{\mathbf{Def}}(s)$ is well-defined, we show that the following formulas hold in the **else** branch:

$$(\exists i \in [1..N] : stayfalse(i, s(\tau_i))) \tag{6}$$

$$t_{j,s(\tau_j)} \in enabled(s). \tag{7}$$

Proofs that these formulas hold and that $\mathrm{PS}_{\mathbf{Def}}(s)$ is persistent in s appear in the Appendix. □

The worst-case time complexity of $\mathrm{PS}_{\mathbf{Def}}$ is the same as $\mathrm{PS}_{\mathbf{Poss}}$ for conjunctions of 1-local predicates, namely $O(N)$. Thus, the overall time complexity of the search is $O(NN_e)$, where N_e is the number of states explored by the algorithm. By the same reasoning as for $\mathrm{PS}_{\mathbf{Poss}}$, N_e is $O(NS)$. Thus, the overall time complexity is $O(N^2S)$.

Example. In evaluation of $c_0 \models \mathbf{Def}\,(p_1 = Y \wedge p_2 = D)$, $t_{1,1}$, $t_{2,1}$, and $t_{2,2}$ are executed. In the resulting state $s_{2,3}$, $t_{2,3}$ is disabled by its guard (executing $t_{2,3}$ would truthify Φ), so $\mathrm{PS}_{\mathbf{Def}}(s_{2,3})$ returns $\{t_{1,2}\}$.

On-line Detection. For on-line detection of $\mathbf{Def}\,\Phi$, detection must be announced as soon as all histories consistent with the known prefix of the computation contain a CGS satisfying Φ. As for on-line detection of $\mathbf{Poss}\,\Phi$, this is easily achieved by modifying the selective search algorithm to explore transitions as they become available, *i.e.*, by replacing the stack with a "to-do set". The algorithm announces that $\mathbf{Def}\,\Phi$ holds whenever the "to-do set" becomes empty.

6 Examples

We implemented our algorithm for detecting $\mathbf{Poss}\,\Phi$ in Java and applied it to two examples.

In the first example, called database partitioning, a database is partitioned among processes 2 through N, while process 1 assigns task to these processes based on the current partition. Each process $i \in [1..N]$ has a variable $partn_i$ containing the current partition. A process $i \in [2..N]$ can suggest a new partition at any time by setting variable chg_i to true and broadcasting a message containing the proposed partition and an appropriate version number. A recipient of

this message accepts the proposed partition if its own version of the partition has a smaller version number or if its own version of the partition has the same version number and was proposed by a process i' with $i' > i$. An invariant I_{db} that should be maintained is: if no process is changing the partition, then all processes agree on the partition.

$$I_{db} = (\bigwedge_{i \in [2..N]} \neg chg_i) \Rightarrow \bigwedge_{i,j \in [1..N], i \neq j} partn_i = partn_j \tag{8}$$

The second example, called primary-secondary, concerns an algorithm designed to ensure that the system always contains a pair of processes that will act together as primary and secondary (*e.g.*, for servicing requests). This is expressed by the invariant

$$I_{pr} = \bigvee_{i,j \in [1..N], i \neq j} isPrimary_i \wedge isSecondary_j \wedge secondary_i = j \wedge primary_j = i. \tag{9}$$

Initially, process 1 is the primary and process 2 is the secondary. At any time, the primary may choose a new primary as its successor by first informing the secondary of its intention, waiting for an acknowledgment, and then multicasting to the other processes a request for volunteers to be the new primary. It chooses the first volunteer whose reply it receives and sends a message to that process stating that it is the new primary. The new primary sends a message to the current secondary which updates its state to reflect the change and then sends a message to the old primary stating that it can stop being the primary. The secondary can choose a new secondary using a similar protocol. The secondary must wait for an acknowledgment from the primary before multicasting the request for volunteers; however, if the secondary receives instead a message that the primary is searching for a successor, the secondary aborts its current attempt to find a successor, waits until it receives a message from the new primary, and then re-starts the protocol.

We implemented a simulator that generates computations of these protocols, and we used state-space search to detect possible violations of the given invariant in those computations, *i.e.*, to detect $\mathbf{Poss}\,\neg I_{db}$ or $\mathbf{Poss}\,\neg I_{pr}$. To apply $\mathrm{PS}_{\mathbf{Poss}}$, we write both predicates as conjunctions. For $\neg I_{db}$, we rewrite the implication $P \Rightarrow Q$ as $\neg P \vee Q$ and then use DeMorgan's Law (applied to the outermost negation and the disjunction). For $\neg I_{pr}$, we simply use DeMorgan's Law. The simulator accepts N and S as arguments and halts when some process has executed $S - 1$ events. Message latencies and other delays are selected randomly from the distribution $1 + \exp(1)$, where $\exp(x)$ is the exponential distribution with mean x. The search optionally uses sleep sets, as described in [God96], as a further optimization. Sleep sets help eliminate redundancy caused by exploring multiple interleavings of independent transitions in a persistent set. Sleep sets are particularly effective for $\mathbf{Poss}\,\Phi$ because, if Φ does not hold in s, then transitions in $\mathrm{PS}_{\mathbf{Poss}}(s)$ are pairwise independent.

Search was done using four levels of optimization: no optimization, persistent sets only, sleep sets only ([God96, Fig. 5.2] with PS = *enabled*), and both persistent sets and sleep sets ([God96, Fig. 5.2] with $\mathrm{PS} = \mathrm{PS}_{\mathbf{Poss}}$).

Data collected by fixing the value of N at 3 or 5 and varying S in the range [2..80] indicate that in all cases, N_T and N_S are linear in S. This is because both examples involve global synchronizations, which ensure that a new local state of any process is not concurrent with any very old local state of any process.

The following table contains measurements for the database partitioning example with $N = 5$ and $S = 80$ and for the primary-secondary example with $N = 9$ and $S = 60$. Using both persistent sets and sleep sets reduced N_T (and, roughly, the running time) by factors of 775 and 72, respectively, for the two examples.

Example	No optimization		Sleep		Persistent		Persis. + Sleep	
	N_T	N_S	N_T	N_S	N_T	N_S	N_T	N_S
database partition	343170	88281	88280	88281	640	545	443	444
primary-secondary	3878663	752035	752034	752035	91874	61773	53585	53586

To help determine the dependence of N_T on N, we graphed $\ln N_T$ *vs.* N and fit a line to it; this corresponds to equations of the form $N_T = e^{aN+b}$. The results are graphed in Figure 4. The dependence on S is linear, so using different values of S in different cases does not affect the dependence on N (*i.e.*, it affects b but not a). In one case, namely, the database partitioning example with both persistent sets and sleep sets, the polynomial $N_T = bN^{1.54}$ yields a better fit than an exponential for the measured region of $N = [3..8]$. The dependence of N_S on N is similar.

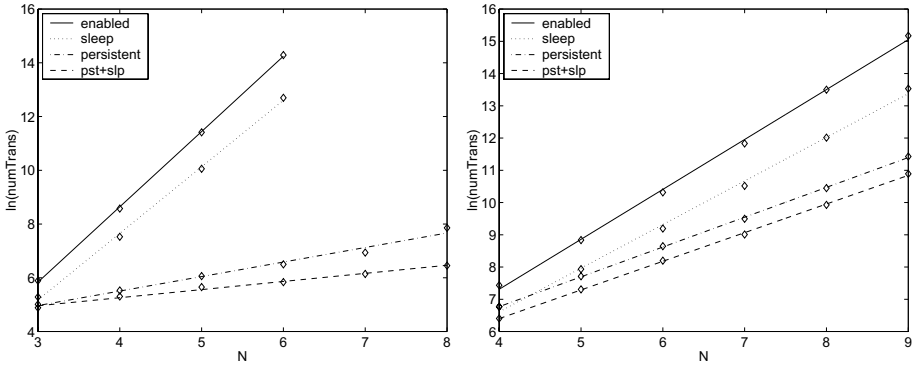

Fig. 4. Datapoints and fitted curves for $\ln N_T$ *vs.* N for all four levels of optimization. Left: Database partitioning example with $S = 25$ for the two searches not using persistent sets and $S = 50$ for the two searches using persistent sets. Right: Primary-secondary example with $S = 60$.

Garg and Waldecker's algorithm for detecting $\mathbf{Poss}\,\Phi$ for conjunctions of 1-local predicates [GW94] is not applicable to the database partitioning example,

because $\neg I_{db}$ contains clauses like $partn_i \neq partn_j$ which are not 1-local. Their algorithm can be applied to the primary-secondary example by putting $\neg I_{pr}$ in disjunctive normal form (DNF) and detecting each disjunct separately. $\neg I_{pr}$ is compactly expressed in conjunctive normal form. Putting $\neg I_{pr}$ in DNF causes an exponential blowup in the size of the formula. This leads to an exponential factor in the time complexity of applying their algorithm to this problem.

References

[AMP98] Rajeev Alur, Ken McMillan, and Doron Peled. Deciding global partial-order properties. In *Proc. 25th International Colloquium on Automata, Languages, and Programming (ICALP)*, volume 1443 of *Lecture Notes in Computer Science*, pages 41–52. Springer-Verlag, 1998.

[AV94] Sridhar Alagar and S. Venkatesan. Techniques to tackle state explosion in global predicate detection. In *Proc. International Conference on Parallel and Distributed Systems*, pages 412–417, December 1994.

[AY99] Rajeev Alur and Mihalis Yannakakis. Model checking of message sequence charts. In *Proc. 10th Int'l. Conference on Concurrency Theory (CONCUR)*, volume 1664 of *Lecture Notes in Computer Science*, pages 114–129, 1999.

[CG98] Craig M. Chase and Vijay K. Garg. Detection of global predicates: Techniques and their limitations. *Distributed Computing*, 11(4):169–189, 1998.

[CM91] Robert Cooper and Keith Marzullo. Consistent detection of global predicates. In *Proc. ACM/ONR Workshop on Parallel and Distributed Debugging*, 1991. ACM SIGPLAN Notices 26(12):167-174, Dec. 1991.

[Fid88] C. Fidge. Timestamps in message-passing systems that preserve the partial ordering. In *Proceedings of the 11th Australian Computer Science Conference*, pages 56–66, 1988.

[FR94] Eddy Fromentin and Michel Raynal. Local states in distributed computations: A few relations and formulas. *Operating Systems Review*, 28(2), April 1994.

[God96] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems*, volume 1032 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[GW94] Vijay K. Garg and Brian Waldecker. Detection of weak unstable predicates in distributed programs. *IEEE Transactions on Parallel and Distributed Systems*, 5(3):299–307, 1994.

[GW96] Vijay K. Garg and Brian Waldecker. Detection of strong unstable predicates in distributed programs. *IEEE Transactions on Parallel and Distributed Systems*, 7(12):1323–1333, 1996.

[JMN95] R. Jegou, R. Medina, and L. Nourine. Linear space algorithm for on-line detection of global predicates. In J. Desel, editor, *Proc. Int'l Workshop on Structures in Concurrency Theory (STRICT '95)*. Springer, 1995.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–564, 1978.

[Mat89] Friedemann Mattern. Virtual time and global states of distributed systems. In M. Corsnard, editor, *Proc. International Workshop on Parallel and Distributed Algorithms*, pages 120–131. North-Holland, 1989.

[MN91] Keith Marzullo and Gil Neiger. Detection of global state predicates. In *Proc. 5th Int'l. Workshop on Distributed Algorithms (WDAG)*, volume 579 of *Lecture Notes in Computer Science*, pages 254–272. Springer, 1991.

[MPS98] Anca Muscholl, Doron Peled, and Zhendong Su. Deciding properties of message sequence charts. In *FoSSaCS '98*, volume 1378 of *Lecture Notes in Computer Science*, pages 226–242, 1998.

[PPH97] Doron Peled, Vaughan R. Pratt, and Gerard J. Holzmann, editors. *Proc. Workshop on Partial Order Methods in Verification*, volume 29 of *DIMACS Series*. American Mathematical Society, 1997.

[SG98] Abraham Silberschatz and Peter B. Galvin. *Operating System Concepts*. Addison Wesley, 5th edition, 1998.

[SUL99] Scott D. Stoller, Leena Unnikrishnan, and Yanhong A. Liu. Efficient detection of global properties in distributed systems using partial-order methods. Technical Report 523, Computer Science Dept., Indiana University, 1999.

[SW89] A. Prasad Sistla and Jennifer Welch. Efficient distributed recovery using message logging. In *Proc. Eighth ACM Symposium on Principles of Distributed Computing*. ACM SIGOPS-SIGACT, 1989.

[TG93] Alexander I. Tomlinson and Vijay K. Garg. Detecting relational global predicates in distributed systems. In *Proc. ACM/ONR Workshop on Parallel and Distributed Debugging*, 1993. ACM SIGPLAN Notices 28(12), December 1993.

[Val97] Antti Valmari. Stubborn set methods for process algebras. In Peled et al. [PPH97], pages 213–231.

[Wal98] Igor Walukiewicz. Difficult configurations - on the complexity of *ltrl*. In *Proc. 25th International Colloquium on Automata, Languages, and Programming (ICALP)*, volume 1443 of *Lecture Notes in Computer Science*, pages 140–151. Springer-Verlag, 1998.

Appendix: Selected Definitions and Proofs

Definition of Independence [God96]. Transitions t_1 and t_2 are *independent* in a state s if

1. Independent transitions can neither disable nor enabled each other, *i.e.*,
   (a) if $t_1 \in \mathit{enabled}(s)$ and $s \xrightarrow{t_1} s'$, then $t_2 \in \mathit{enabled}(s)$ iff $t_2 \in \mathit{enabled}(s')$;
   (b) condition (a) with t_1 and t_2 interchanged holds; and
2. Enabled independent transitions commute, *i.e.*, if $\{t_1, t_2\} \subseteq \mathit{enabled}(s)$, then there is a unique state s' such that $s \xrightarrow{t_1} s_1 \xrightarrow{t_2} s'$ and $s \xrightarrow{t_2} s_2 \xrightarrow{t_1} s'$.

Proof of Theorem 1: It suffices to show that for each transition t in T, t is independent with t_n in s_n. Φ is false in s, so $t_0 \notin \mathit{enabled}(s)$, so t is $t_{i,k}$, as defined in (1), for some i and k. Note that $t_n \neq t_0$, because by hypothesis, Φ is false in s_n. The transitions of each process occur in the order they are numbered, and $t_{i,k} \in \mathit{enabled}(s)$, so the next transition of process i that is executed after state s is $t_{i,k}$; in other words, from state s, no transition of process i can occur before $t_{i,k}$ does. Since $t_{i,k} \in T$ and $(\forall i \in [1..n] : t_i \notin T)$, it follows that t_n is a transition of some process i' with $i' \neq i \wedge i' \neq 0$. From the structure of the system, it is easy to show that, once $t_{i,k}$ has become enabled, such a transition t_n cannot enable or disable $t_{i,k}$ and commutes with $t_{i,k}$ when both are enabled. Thus, $t_{i,k}$ and t_n are independent in s_n. □

Proof of Theorem 2: First we show that (6) holds in the **else** branch. In that branch, $enabled(s) \neq \emptyset$, so $enabled(s)$ contains some transition $t_{j,s(\tau_j)}$. Let $s \stackrel{t_{j,s(\tau_j)}}{\rightarrow} s'$. Suppose ϕ_j is true in s'. The guard of $t_{j,s(\tau_j)}$ implies that Φ is false in s', so there exists $i \neq j$ such that ϕ_i is false in s'. A transition of process j does not change the local state of process i, so $s(p_i) = s'(p_i)$, so ϕ_i is false in s, so $stayfalse(i, s(\tau_i))$ holds.

Suppose ϕ_j is false in s'. If ϕ_j is false in s, then $stayfalse(j, s(\tau_j))$ holds. Otherwise, suppose ϕ_j is true in s. Since s is reachable, Φ is false in s, so there exists $i \neq j$ such that ϕ_i is false in s. Note that $s(p_i) = s'(p_i)$, so ϕ_i is false in s', so $stayfalse(i, s(\tau_i))$ holds.

Now we show that (7) holds in the **else** branch. By (6), $stayfalse(i, s(\tau_i))$ holds for some i. Let j be the node with no outedges that was selected by the algorithm. If suffices to show that executing $t_{j,s(\tau_j)}$ from state s would not cause Φ to become true. If $i = j$, then this follows immediately from the second conjunct in $stayfalse(i, s(\tau_i))$. If $i \neq j$, then this follows immediately from the first conjunct in $stayfalse(i, s(\tau_i))$ and the fact that $t_{j,s(\tau_j)}$ does not change the local state of process i. Note that, in both cases, ϕ_i is false in s'.

Finally, we show that $\mathrm{PS}_{\mathbf{Def}}(s)$ is persistent in s. This is trivial if $enabled(s) = \emptyset$. Suppose $enabled(s) \neq \emptyset$, so $stayfalse(i, s(\tau_i))$ holds. Let $\{t_{j,s(\tau_j)}\} = \mathrm{PS}_{\mathbf{Def}}(s)$. Let $s_1 \stackrel{t_1}{\rightarrow} s_2 \stackrel{t_2}{\rightarrow} s_3 \cdots \stackrel{t_{n-1}}{\rightarrow} s_n \stackrel{t_n}{\rightarrow} s_{n+1}$ such that $s_1 = s$ and $(\forall k \in [1..n] : t_k \neq t)$. It suffices to show that t_n and $t_{j,s(\tau_j)}$ are independent in s_n. As noted at the end of the previous paragraph, executing $t_{j,s(\tau_j)}$ in s leaves ϕ_i false. $t_1, \ldots, t_n$ are not transitions of process i, because the next transition of process i cannot occur before $t_{j,s(\tau_j)}$ occurs (this follows from the choice of j and the definition of wait-for graph). Thus,

$$(\forall k \in [1..n+1] : s_k(p_i) = s(p_i) \ \wedge \ \text{executing } t_{j,s(\tau_j)} \text{ in } s_k \text{ leaves } \phi_i \text{ false}). \quad (10)$$

Consider the requirements in the definition of independence.

(1a) Suppose $t_{j,s(\tau_j)} \in enabled(s_n)$. Let $s_n \stackrel{t_{j,s(\tau_j)}}{\rightarrow} s'$. By hypothesis, $t_n \in enabled(s_n)$, so we need to show that $t_n \in enabled(s')$. Since t_n is enabled in s_n, it suffices to show that executing t_n in s' leaves Φ false. By (10), executing $t_{j,s(\tau_j)}$ in s_n leaves ϕ_i false, and t_n is not a transition of process i, so executing t_n in s' leaves ϕ_i false.

(1b) Suppose $t_n \in enabled(s_n)$. Recall that $s_n \stackrel{t_n}{\rightarrow} s_{n+1}$. We need to show that $t_{j,s(\tau_j)} \in enabled(s_n)$ iff $t_{j,s(\tau_j)} \in enabled(s_{n+1})$. $t_{j,s(\tau_j)}$ is enabled in s, and by (10), executing $t_{j,s(\tau_j)}$ in s_n or in s_{n+1} leaves ϕ_i and hence Φ false. It follows that $t_{j,s(\tau_j)}$ is enabled in both s_n and s_{n+1}.

(2) It is easy to show that $t_{j,s(\tau_j)}$ and t_n commute.

Thus, t_n and $t_{j,s(\tau_j)}$ are independent in s_n, and $\mathrm{PS}_{\mathbf{Def}}(s)$ is persistent in s. □