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

Scott D. Stoller   Leena Unnikrishnan   Yanhong A. Liu

15 July 2000

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. The resulting detection algorithms handle larger classes of predicates and thus are more general than two special-purpose detection algorithms by Garg and Waldecker, which detect **Poss** Φ and **Def** Φ efficiently for a restricted but important class of predicates. Furthermore, our algorithm for **Poss** Φ achieves the same worst-case asymptotic time complexity as Garg and Waldecker's special-purpose algorithm for **Poss** Φ. 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 sequence of global states through which the system passed. This leads to an obvious difficulty for detecting whether a global property held.

*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}/

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 aforementioned algorithm of Garg and Waldecker for $\mathbf{Poss}\,\Phi$, 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 $\mathbf{Poss}\,\Phi$ for conjunctions of 1-local predicates with time complexity $O(N^2S)$. In some non-worst cases, Garg and Waldecker's algorithm 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 Section 8.

Our algorithms for detecting **Poss** Φ and **Def** Φ handle larger classes of predicates and thus are more general than Garg and Waldecker's algorithms. Furthermore, our method for **Poss** Φ 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 classes of systems, although our methods for **Poss** Φ and **Def** Φ have 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 **Poss** Φ 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 specialized algorithms $\mathrm{PS}_{\mathbf{Poss}}$ and $\mathrm{PS}_{\mathbf{Def}}$ for computing persistent sets for detection of **Poss** Φ and **Def** Φ, 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 **Poss** Φ, 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 **Poss** Φ. When CSSA (or any of the other algorithms in [God96]) is used for detecting **Def** Φ, it is ineffective.

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 $\mathrm{ISTL}^{\diamond}$. **Poss** Φ is expressible in $\mathrm{ISTL}^{\diamond}$ as $\exists\Diamond\Phi$. **Def** Φ 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].

Using known partial-order methods for temporal-logic model checking [God96, chapter 7], it should be possible to use persistent-set selective search as the basis of new and efficient algorithms for detecting temporal global properties in asynchronous distributed systems, such as the "behavioral patterns" of Babaoğlu and Raynal [BR94].

An interesting open question is how the performance of persistent-set selective search compares with that of symbolic methods (*e.g.*, [SL98, Hel99]) for this class of problems.

Section 2 provides background on property detection. Section 3 provides background on partial-order methods. Sections 4 and 6 present our algorithms for detecting **Poss** Φ and **Def** Φ, respectively. Section 5 describes an optimized algorithm for detecting **Poss** Φ that can explore sequences of transitions in a single step. Section 7 demonstrates the effectiveness of our approach on two examples. Section 8 discusses other enhancements and optimizations. Section 9 compares $\mathrm{PS}_{\mathbf{Poss}}$ and $\mathrm{PS}_{\mathbf{Def}}$ to general-purpose algorithms for computing persistent sets.

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 i has a distinguished variable vt such that for each local state s in computation c, $s(vt)$ is a *vector timestamp* [Mat89], *i.e.*, an array of N natural numbers such that for each j, if the number k of local states of process j that happened-before s in c is zero and $i \neq j$, then $s(vt)[j] = 0$, otherwise $s(vt)[j] = k + 1$. Vector timestamps capture the happened-before relation. Specifically, for all local states s_1 and s_2, $s_1 \rightarrow s_2$ iff $s_1(vt) \neq s_2(vt) \wedge (\forall i \in [1..N] : s_1(vt)[i] \leq s_2(vt)[i])$. 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.

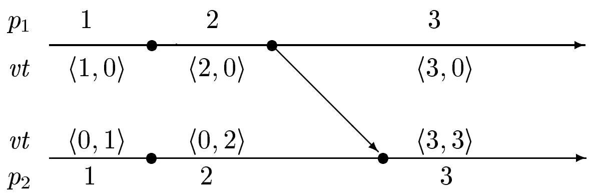

Figure 1: Computation c_0.

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 **Poss** Φ, 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: $s_1 \preceq_G s_2 \;=\; (\forall i \in [1..N] : s_1[i] = s_2[i] \;\vee\; s_1[i] \to s_2[i])$. 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 $\preceq_G$: (*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 **Def** Φ, 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 $\{s_{1,1}, s_{2,1}, s_{1,2}, s_{2,2}, s_{3,1}, s_{3,2}, s_{3,3}\}$. Some properties of this computation are: $c_0 \models \mathbf{Poss}\,(p_1 = 2 \wedge p_2 = 2)$, $c_0 \models \mathbf{Def}\,(p_1 + p_2 = 2)$, $c_0 \not\models \mathbf{Def}\,(p_1 = 2 \wedge p_2 = 2)$, and $c_0 \not\models \mathbf{Poss}\,(p_1 = 1 \wedge p_2 = 3)$.

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"

[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 $x \neq y \;\wedge\; x \prec y \;\wedge\; \neg(\exists\, z \in S \setminus \{x, y\} : x \prec z \wedge z \prec y)$.

has different meanings in the two models. Sections 4 and 6 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. For convenience, assume $P_i \cap P_j = \emptyset$ for $i \neq j$.
- $\mathcal{O}$ 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.

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

```
stack := empty;
H := empty;
push initial state onto stack;
while (stack is not empty)
    pop a state s from stack;
    if (s is not in H) then
        insert s in H;
        for all t in PS(s)
            s' := the state that results from executing t in s;
            push s' onto stack
```

Figure 2: Algorithm PSSS (Persistent-Set Selective Search). PS(s) returns a persistent set.

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 $T \subseteq \mathit{enabled}(s)$ 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). Pseudo-code appears in Figure 2. Note that $\mathit{enabled}(s)$ 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 $c \models \textbf{Poss}\,\Phi$ 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 $\ell_{0,nd}$ (this ensures that transition t_0 disables all transitions), 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

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

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*(ϕ), 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); this is the simplest choice that satisfies the definition. 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 $s_1 \overset{t_1}{\to} s_2 \overset{t_2}{\to} \cdots s_n \overset{t_n}{\to} s_{n+1}$, 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*(ϕ) is executed. Note that the next transition of process i must be $t_{i,s(\tau_i)}$. Thus, for each process i in *supp*(ϕ), 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 3.

The wait-for graph can be incrementally maintained in $O(N)$ time per transition, because a transition $t_{i,k}$ can affect only the $O(N)$ edges incident on the node for process i. 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 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 by advancing process i past its k'th local state. $|\mathcal{T}|$ is $O(NS)$, so N_e is also $O(NS)$, so the overall time complexity is $O(N^2S)$.

$\mathrm{PS}_{\mathbf{Poss}}(s)$
  if (Φ holds in s) $\vee$ $enabled(s) = \emptyset$ **then return** $enabled(s)$
  else choose some conjunct ϕ of Φ such that ϕ is false in s;
    return followWF(s, $supp(\phi)$)

followWF(s,$procs$)
  $T := \emptyset$;
  for all i in $procs$ such that $s(\tau_i) < |c[i]|$
    start at node i in $WF(s)$;
    follow any path until a node j with no outedges is reached;
    insert $t_{j,s(\tau_j)}$ in T;
  return T

Figure 3: Algorithm $\mathrm{PS}_{\mathbf{Poss}}(s)$.

Example. Consider evaluation of $c_0 \models \mathbf{Poss}\,(p_1 = 2 \wedge p_2 = 3)$. Recall that this is false. For this example, we resolve the non-determinism in $\mathrm{PS}_{\mathbf{Poss}}$ by taking the lowest-numbered choice. $\mathrm{PS}_{\mathbf{Poss}}(s_{1,1})$ returns $\{t_{1,1}\}$, although both processes are enabled in $s_{1,1}$. Similarly, $\mathrm{PS}_{\mathbf{Poss}}(s_{2,1})$ returns $\{t_{2,1}\}$, although both processes are enabled in $s_{1,1}$. Thus, the selective search avoids exploring $s_{1,2}$, $s_{3,1}$, and teh four transitions incident on them.

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 $\Theta(S^3)$, because their algorithm does not exploit the structure of the predicate.

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 Selective Search using Sequences of Transitions

When PSSS is used to detect **Poss**, the following optimization can be used to reduce the number of explored states and transitions. In the **else** branch of $\mathrm{PS}_{\mathbf{Poss}}$ (in Figure 3), if the next transition of process i is not enabled, then a path p from i to a node j with no outedges in $WF(s)$ is found, and j is added to T. If $j \notin supp(\phi)$, then ϕ and hence Φ cannot be truthified merely by executing the next transition of process j. More generally, ϕ cannot hold until some process in $supp(\phi)$ has advanced. This suggests the following optimization. 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. This sequence advances other processes just enough to enable the next transition of process i. If process i is waiting for some other process in $supp(\phi)$, then insert in T a minimum-length sequence w of transitions that ends with a transition of some process in $supp(\phi)$ and that does not contain any other transitions of processes in $supp(\phi)$.

To see that exploring the entire sequence w in a single step (and therefore not exploring any other transitions out of the intermediate states) is a correct optimization, observe that Φ does not hold in the intermediate states. Furthermore, suppose there is a state s' such that $s'(\Phi)$ holds and s' is reachable from one of the intermediate states by a sequence w' of transitions. Then $s'(\phi)$ holds, so w' must contain a transition by a process in $supp(\phi)$, so w' must contain all the transitions in some sequence w in T. Now we show that the transitions in w can be moved to the front of w' by repeated swapping of adjacent transitions that commute. By definition of w, the transitions in w can be executed directly from s (*i.e.*, without executing any other transitions first). The structure of the transitions implies that concurrently enabled transitions in w'—except t_0, if present—commute. If t_0 appears in w', it must be the last transition in w', so all transitions in w can be moved to the front of w' without a need to commute them with t_0. Thus, s' is reachable from s by a sequence of transitions that has w as a prefix.

Example. Consider the computation c_1 illustrated in Figure 4. Processes 1 and 3 each have S distinct local states; process 2 has a small constant number of distinct local states. Suppose we want to evaluate $c_1 \models \mathbf{Poss}\, \phi(p_2, p_3)$. If the predicate remains false during the search, then the algorithm of Section 4 considers all interleavings of transitions of processes 1 and 3, so the worst-case time complexity is $\Theta(S^2)$. With the optimization presented in this section, all transitions of process 1 are executed in a single step (together with a transition of process 2), so the worst-case time complexity is $\Theta(S)$.

6 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

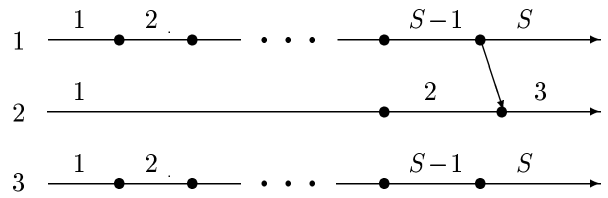

Figure 4: Computation c_1.

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 $s_{\rm init}$ 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 process 0 has control points $\ell_{0,nv}$ (mnemonic for "no violation of $\mathbf{Def}\,\Phi$ detected") and $\ell_{0,v}$ ("violation of $\mathbf{Def}\,\Phi$ detected"). Thus, $\mathcal{P} = \bigcup_{i=1..N}\{P_i\}$, where $P_0 = \{\ell_{0,nv}, \ell_{0,v}\}$ and for $i > 0$, $P_i = \bigcup_{k=1..|c[i]|}\{\ell_{i,k}\}$. The shared variables are the same as in Section 4, plus a boolean shared variable h that keeps track of whether a global state satisfying Φ has been encountered. Thus, $\mathcal{O} = \{h\} \cup \bigcup_{i=1..N}\{p_i, \tau_i\}$. The initial state is the same as in Section 4, except the control point for process 0 is $l_{0,nv}$ and the initial value of h equals the value of Φ in the initial state of c. The transitions are similar to those in Section 4, except the transition t_0 executes, indicating a violation of $\mathbf{Def}\,\Phi$, if the final state of c is reachable with $h = \text{false}$ (*i.e.*, without passing through a state satisfying Φ). The transitions are $\mathcal{T} = \{t_0\} \cup \bigcup_{i=1..N,k=1..|c[i]|-1}\{t_{i,k}\}$, where $t_0 = \langle\{l_{0,nv}\}, \neg h \wedge (\forall i \in [1..N] : \tau_i = |c[i]|), \text{skip}, \{l_{0,v}\}\rangle$ and

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

Note that every reachable state containing $l_{0,v}$ is a deadlock. It is easy to show that $c \not\models \mathbf{Def}\,\Phi$ iff a deadlock containing $l_{0,v}$ is reachable.

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

$$\begin{array}{lcl} t_{1,1} & = & \langle\{\ell_{1,1}\},\ \tau_1 = 1\ \wedge\ \tau_2 \geq 1,\ p_1 := 2;\ \tau_1 := 2;\ h := h \vee \Phi(p_1, p_2),\ \{\ell_{1,2}\}\rangle \\ t_{1,2} & = & \langle\{\ell_{1,2}\},\ \tau_1 = 2\ \wedge\ \tau_2 \geq 1,\ p_1 := 3;\ \tau_1 := 3;\ h := h \vee \Phi(p_1, p_2),\ \{\ell_{1,3}\}\rangle \\ t_{2,1} & = & \langle\{\ell_{2,1}\},\ \tau_2 = 1\ \wedge\ \tau_1 \geq 1,\ p_2 := 2;\ \tau_2 := 2;\ h := h \vee \Phi(p_1, p_2),\ \{\ell_{2,2}\}\rangle \\ t_{2,2} & = & \langle\{\ell_{2,2}\},\ \tau_2 = 2\ \wedge\ \tau_1 \geq 3,\ p_2 := 3;\ \tau_2 := 3;\ h := h \vee \Phi(p_1, p_2),\ \{\ell_{2,3}\}\rangle \\ t_0 & = & \langle\{l_{0,nv}\},\ \neg h\ \wedge\ \tau_1 = 3\ \wedge\ \tau_2 = 3,\ \text{skip},\ \{l_{0,v}\}\rangle \end{array} \tag{4}$$

Without loss of generality, we write Φ as a conjunction: $\Phi = \bigwedge_{i=1..n} \phi_i$, with $n \geq 1$. $\mathit{active}(t)$ is the set of processes whose control points appear in transition t. For a set T of transitions, $\mathit{active}(T) = \bigcup_{t \in T} \mathit{active}(t)$. For a 1-local predicate ϕ that depends on the state of process i, $\mathit{nextTrue}(k, \phi)$ is the index of the "next" local state of process i satisfying ϕ, *i.e.*, the smallest $k_1 \geq k$ such that $\phi(c[i][k_1]) = \text{true}$; this can be pre-computed for all k and ϕ in $O(SN)$ time. For $t \in \mathit{enabled}(s)$, $\mathit{exec}(s,t)$ is the state obtained by executing t from s, *i.e.*, $s \xrightarrow{t} \mathit{exec}(s,t)$. For a 1-local predicate ϕ', a state s such that $s \not\models \phi'$, and a set T of transitions, $\mathit{falseUntilEnd}(\phi', s, T)$ ensures that ϕ' remains false for all sequences of transitions starting from s and staying outside T, except possibly in the last state of such a sequence (this is discussed in more detail in the proof of Theorem 2).

$$\begin{aligned} &\mathit{falseUntilEnd}(\phi', s, T) = \text{let } \{j\} = \mathit{supp}(\phi') \text{ in} \\ &\quad (\exists i \in \mathit{active}(T) : c[i][s(\tau_i)] \rightarrow c[j][\mathit{nextTrue}(s(\tau_j), \phi') + 1]) \\ &\quad \wedge\ (\forall i_1 \in [1..N] \setminus (\mathit{supp}(\phi) \cup \mathit{active}(T)) : s(\tau_{i_1}) = |c[i_1]| \\ &\qquad \vee\ (\exists i_2 \in \mathit{active}(T) : i_2 \text{ is reachable from } i_1 \text{ in } \mathit{WF}(s))) \end{aligned} \tag{5}$$

Pseudo-code for $\mathrm{PS}_{\mathbf{Def}}$ appears in Figure 5. The underlying idea in cases $(c) - (e)$ is that, for a state s with $s(h) = \text{false}$, $T \subseteq \mathit{enabled}(s)$ is persistent in s if Φ remains false for all sequences of transitions starting from s and staying outside T except that the last transition might be in T. This idea is evident in the proof of Theorem refthm:def. $\mathrm{PS}_{\mathbf{Def}}(s)$ might return a singleton set if $s(h) = \text{true}$; however, it is easy to see that exploring transitions from such states is unnecessary.

Theorem 2. $\mathrm{PS}_{\mathbf{Def}}(s)$ is persistent in s.

Proof: See Appendix. ∎

For conjunctions of 1-local predicates, the worst-case time complexity of $\mathrm{PS}_{\mathbf{Def}}$ is the same as $\mathrm{PS}_{\mathbf{Poss}}$, namely $O(N)$, assuming the wait-for graph is incrementally maintained and *nextTrue* is pre-computed. Thus, the overall time complexity of the search is $O(NN_e)$, where N_e is the number of states explored by the algorithm. $\mathrm{PS}_{\mathbf{Def}}(s)$ sometimes returns $\mathit{enabled}(s)$, so N_e is $\Theta(S^N)$ in the worst-case. In many cases, though, $\mathrm{PS}_{\mathbf{Def}}(s)$ returns a proper subset of $\mathit{enabled}(s)$, and the selective search explores many fewer states than Cooper and Marzullo's algorithm.

$\mathrm{PS}_{\mathbf{Def}}(s)$

if $enabled(s) = \emptyset$ **then return** $\emptyset$ (a)

else if $s(h) = \text{true}$ **then**

choose some $t \in enabled(s)$;

return $\{t\}$ (b)

else choose some conjunct ϕ that is false in s;

$T := \text{followWF}(s, supp(\phi))$;

if $(\forall t \in T : exec(s,t) \not\models \phi)$ **then return** T (c)

else if there is a conjunct ϕ' such that $s \not\models \phi' \wedge (supp(\phi) \cap supp(\phi') = \emptyset)$ **then**

if (ϕ' is 1-local) $\wedge\, falseUntilEnd(\phi', s, T)$ **then return** T (d)

else return $T \cup \text{followWF}(s, \phi')$ (e)

else return $enabled(s)$ (f)

Figure 5: Algorithm $\mathrm{PS}_{\mathbf{Def}}(s)$.

Example. Consider evaluation of $c_0 \models \mathbf{Def}\,(p_1 = 2 \wedge p_2 = 2)$. Recall that this is false. $\mathrm{PS}_{\mathbf{Def}}(s_{1,1})$ returns $\{t_{1,1}\}$, although both processes are enabled in $s_{1,1}$. Thus, the selective search avoids exploring $s_{1,2}$ and the two transitions incident on it. Also, h becomes true in $s_{2,2}$, so it is unnecessary to explore any transitions from that state.

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 h is true in all states in the "to-do set". When using $\mathrm{PS}_{\mathbf{Def}}$ on-line, *falseUntilEnd* and *nextTrue* cannot be evaluated if insufficiently many local states of process j have arrived; in that case, simply take *falseUntilEnd* to be false.

7 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.

$$I_{db} = (\bigwedge_{i \in [2..N]} \neg chg_i) \Rightarrow \bigwedge_{i,j \in [1..N], i \neq j} partn_i = partn_j \tag{6}$$

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

$$I_{pr} = \bigvee_{i,j \in [1..N], i \neq j} isPrimary_i \wedge isSecondary_j \wedge secondary_i = j \wedge primary_j = i. \tag{7}$$

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. This prevents the primary and secondary from trying to choose successors concurrently.

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 (*e.g.*, how long to wait before suggesting a new partition or looking for a successor) 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 (PSSS with PS = *enabled*), persistent sets only (PSSS with PS = $\mathrm{PS}_{\mathbf{Poss}}$), sleep sets only ([God96, Fig. 5.2] with PS = *enabled*), and both persistent sets and sleep sets ([God96, Fig. 5.2] with PS = $\mathrm{PS}_{\mathbf{Poss}}$). Most of the results below represent averages over 100 computations; for large values of N or S, averages over 5 computations were used; the results were consistent enough that using more computations seemed unnecessary. Let N_T and N_S denote the number of explored transitions and states, respectively.

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 (*i.e.*, for all four levels of optimization for both examples), 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 (using polyfit and polyval in Matlab); this corresponds to equations of the form $N_T = e^{aN+b}$. The results appear in the following table and are graphed in Figure 6. The table also includes the ranges of values of N and S used for the fits. 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).

Example	No optimization	Sleep	Persistent	Persis. + Sleep
database partition	$N_T = e^{2.80N-2.57}$ $S = 25, N = [3..6]$	$N_T = e^{2.48N-2.26}$ $S = 25, N = [3..6]$	$N_T = e^{0.540N+3.35}$ $S = 50, N = [3..8]$	$N_T = e^{0.301N+4.06}$ $S = 50, N = [3..8]$
primary-secondary	$N_T = e^{1.55N+1.12}$ $S = 60, N = [4..9]$	$N_T = e^{1.35N+1.20}$ $S = 60, N = [4..9]$	$N_T = e^{0.925N+3.07}$ $S = 60, N = [4..9]$	$N_T = e^{0.888N+2.85}$ $S = 60, N = [4..9]$

To determine whether a polynomial form would provide a better fit than the exponential form used above, we also graphed $\ln N_T$ *vs.* $\ln N$ and fit a line to it. The exponential form produced a better fit in all cases except one: for the database partitioning example with both persistent sets and sleep sets, $N_T = N^{1.54}e^{3.16}$, shown in Figure 7, fits better than $N_T = e^{0.301N+4.06}$, shown in Figure 6 (goodness of fit, measured by sum of squares of differences, is 0.0155 *vs.* 0.0173). Thus, within the measured region, the low-order polynomial form provides a better fit than the exponential form. The results for N_S are similar to those for N_T. Specifically, in the measured region, the dependence on N_S on N fits better to an exponential form than a polynomial form in all cases except one, namely, the database partitioning example with both persistent sets and sleep sets.

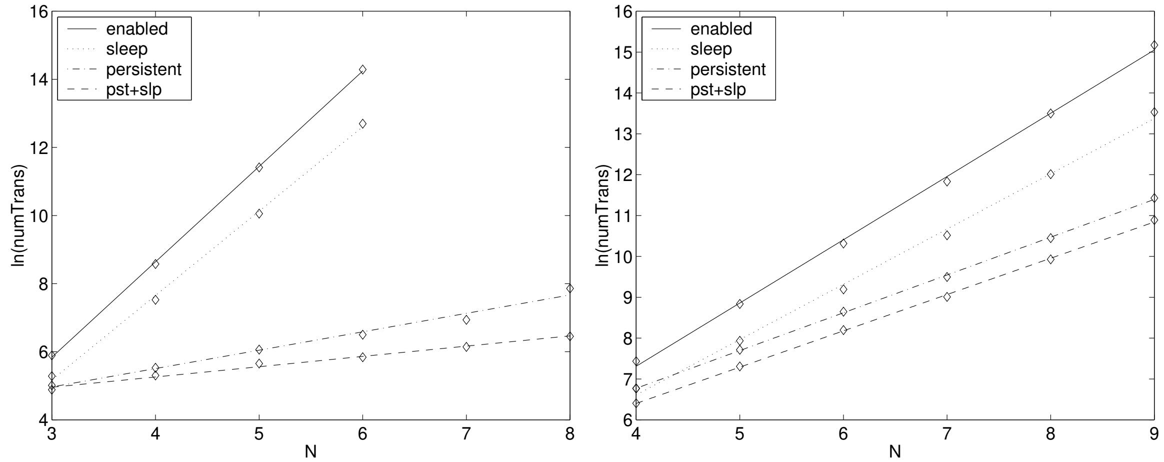

Figure 6: Datapoints and fitted curves for $\ln N_T$ *vs.* N for all four levels of optimization. Left: Database partitioning example with $S = 25$ for the two searches not using persistent sets and $S = 50$ for the two searches using persistent sets. Right: Primary-secondary example with $S = 60$.

In all cases, the time required for the search is directly proportional to the number of explored transitions, so we generally report only the latter. The constant of proportionality is somewhat larger for the optimized searches. For example, consider the primary-secondary example with $N = 9$ and $S = 60$. With no optimizations, 3878663 transitions were explored in 254 sec; with persistent sets and sleep sets, 53585 transitions were explored in 5.58 sec. Thus, the detection algorithms explored approximately 15300 transitions/sec and 9600 transitions/sec, respectively, in these two cases.

A *state match* occurs when an explored transition leads to a previously visited global state. The number of state matches equals $N_T - (N_S - 1)$, since every transition leads to a new global state or a previously visited state other than the start state. Sleep sets completely eliminated state matches.

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.

$$\neg I_{pr} = \bigwedge_{i,j \in [1..N], i \neq j} \neg isPrimary_i \vee \neg isSecondary_j \vee secondary_i \neq j \vee primary_j \neq i \qquad (8)$$

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.

Stoller and Schneider's algorithm for detecting $\mathbf{Poss}\,\Phi$ [SS95] offers no benefit for formulas with

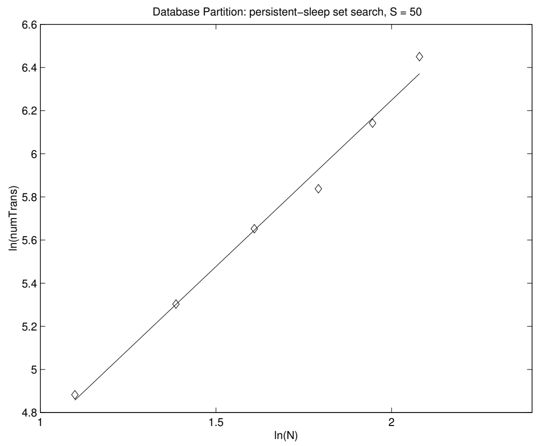

Figure 7: Datapoints and fitted curves for $\ln N_T$ *vs.* $\ln N$ for search using persistent sets and sleep sets for database partitioning example with $S = 50$.

the logical structure of $\neg I_{db}$ or $\neg I_{pr}$, because the minimum-sized fixed sets for such formulas have size $N - 1$.

8 Discussion

Weak Vector Clocks. An important optimization, applicable to detecting **Poss** and **Def**, is to use weak vector clocks, as described by Marzullo and Neiger [MN91]. Standard vector clocks [Mat89], as sketched in Section 2, are updated by every event. Weak vector clocks 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. A local state of process i is appended to $c[i]$ only when the (weak or standard) vector clock of process i changes, so using weak vector clocks can decrease S. Let S_w denote that decreased value. Garg and Waldecker use a different technique for reducing the number of local states considered by their algorithms. Their algorithm for **Poss** has worst-case time complexity $O(N^2 S_s)$, where S_s is the maximum number of send events by a single process. Their algorithm for **Def** has worst-case time complexity $O(N^2 S_r)$, where S_r is the maximum number of receive events by a single process. For some classes of computations, S_r or S_s are smaller than S_w by a factor of N. For some classes of computations, S_s, S_r, and S_w are asymptotically equal.

Possibility Detection Decomposition. The Possibility Detection Decomposition Algorithm (PDDA) of Stoller and Schneider [SS95] detects **Poss** Φ for arbitrary predicates Φ, is as fast as Garg and Waldecker's algorithm when Φ is a conjunction of 1-local predicates, is asymptotically faster than Cooper and Marzullo's algorithm for a general class of predicates, and is never asymptotically slower than Cooper and Marzullo's algorithm. PDDA uses as a subroutine an algorithm for detecting $\mathbf{Poss}\,\Phi$ for Φ a conjunction of 1-local predicates. Naturally, PSSS can be used as that subroutine. Furthermore, the fundamental idea underlying PDDA can be viewed as a specialized partial-order method. Roughly speaking, the idea is to choose an appropriate set F of processes, called a *fixed set*, and explore only sequences of transitions in which processes in F advance first and then processes outside F advance (actually, during the first phase, the processes outside F might need to advance, too, but just enough to enable transitions of processes in F; this is independent of the predicate being detected). This avoids exploring many interleavings of the transitions of processes in F with transitions of processes outside F. Persistent sets and sleep sets do not seem to capture this optimization. One direction for future work is to see whether this specialized partial-order method can be generalized to apply to other state-space search problems.

9 Comparison to General-Purpose Persistent Set Algorithms

This section compares $\mathrm{PS}_{\mathbf{Poss}}$ and $\mathrm{PS}_{\mathbf{Def}}$ with the general-purpose persistent set algorithms in [God96]. Familiarity with [God96, Chapter 4] is assumed.

9.1 Persistent Sets for Poss

One of the most sophisticated general-purpose algorithms for computing persistent sets is the conditional stubborn set algorithm (CSSA) [God96, Section 4.7], which is based on Valmari's work on stubborn sets [Val97]. We consider two ways of using CSSA instead of $\mathrm{PS}_{\mathbf{Poss}}$ for detecting $\mathbf{Poss}\,\Phi$. With the first way, CSSA is ineffective, returning all enabled transitions. With the second way, CSSA is slower than $\mathrm{PS}_{\mathbf{Poss}}$ by a factor of S (and possibly by some factors of N, which we did not analyze in detail) in the worst case.

9.1.1 Comparison to CSSA Based on Read and Write Operations

We take the operations on all shared variables to be read and write. CSSA is parameterized by a binary relation $\triangleright_s$ on operations [God96, p. 65], where $op \triangleright_s op'$ is read as "op' might be the first operation to interfere with op from s". We adopt the usual notion of interference between reads and writes: if op is a read from a shared variable v and op' is a write to v, then $op \triangleright_s op'$. Consider applying CSSA in a state s. In step 1, we choose a transition t_e that is enabled in s, and take $T_s = \{t_e\}$. We execute step 2 for the first time, with $t = t_e$. Since t_e is enabled in s, we execute step 2(b). For a transition $\langle L_1, G, C, L_2\rangle$, let $\mathit{pre}(\langle L_1, G, C, L_2\rangle) = L_1$. Note that for all transitions t_1 and t_2 with $t_1 \neq t_2$, $\mathit{pre}(t_1) \cap \mathit{pre}(t_2) = \{\ell_{0,nd}\}$. Thus, for all transitions $t' \neq t_e$, t' and t_e are in conflict [God96, p. 44], so after step 2(b)i, $T_s = \mathcal{T}$. Thus, CSSA returns $\mathit{enabled}(s)$.

We can eliminate this problem by re-defining the transitions. Specifically, we eliminate both occurrences of $\ell_{0,nd}$ from the definition (1) of $t_{i,k}$. Now, $c \models \mathbf{Poss}\,\Phi$ iff local state $\ell_{0,d}$ is reachable, *i.e.*, iff a state (which is not necessarily a deadlock) containing $\ell_{0,d}$ is reachable. Selective search using persistent sets and sleep sets can be used to test reachability of a local state, provided the dependency relation is weakly uniform [God96, Section 6.3]. Consider applying CSSA in a state s. In step 1, we choose a transition t_e that is enabled in s, and take $T_s = \{t_e\}$. We execute step 2 for the first time, with $t = t_e$. Since t_e is enabled in s, we execute step 2(b). Step 2(b)i adds no transitions to T_s, because no transition $t' \neq t_e$ is in conflict with t_e. For each $j \notin \{0, i\}$, let t'_j be the next transition of process j, *i.e.*, $t'_j = t_{j,s(\tau_j)}$. The command of t'_j writes to τ_j, and the guard of t reads from τ_j, and the former operation might be the first to interfere with the latter from s, so step 2(b)ii adds t'_j to T_s. Thus, CSSA returns $enabled(s)$.

One might hope to eliminate this problem by classifying the write operations $v := a$ and $v := a'$ with $a \neq a'$ as different operations on v. This allows a more refined relation $\triangleright_s$ to be introduced, since from a given state s, only certain values can be the first value assigned to a given variable in a sequence of transitions starting from s. However, this refinement alone does not actually have any effect on this problem. A deeper refinement of the operations does have an effect, as described next.

9.1.2 Comparison to CSSA Based on Computation-Specific Operations

In this section, we customize the operations on shared variables $\tau_1, \ldots, \tau_N$ according to the computation. Specifically, each conjunct in the guard of a transition $t_{i,k}$ is considered to be a boolean-valued operation on the variable that it accesses (each conjunct accesses exactly one variable). Operations of the form $c[j][\tau_j](vt)[j] \geq c[i][k+1](vt)[j]$ are computation-specific, *i.e.*, they depend on the computation c. The operations on the variables variables $p_1, \ldots, p_N$ are taken to be reads and writes, as in Section 9.1.1. Write operations $v := a$ and $v := a'$ with $a \neq a'$ are considered to be different operations on v.

The dependency relations on the operations of each object are as follows. Introduce the following abbreviations for operations on τ_j :

$$
\begin{aligned}
assign(j,k) &= \tau_j := k+1 && (9)\\
test(i,j,k) &= c[j][\tau_j](vt)[j] \geq c[i][k+1](vt)[j] && (10)
\end{aligned}
$$

The conditional dependency relation for τ_j is the smallest symmetric (in the first two arguments) ternary relation such that (free variables are implicitly universally quantified)

$$
\begin{aligned}
k \neq k' &\Rightarrow \langle assign(j,k), assign(j,k'), v\rangle \in D_{\tau_j} && (11)\\
test(i,j,k)[\tau_j := v] \neq test(i,j,k)[\tau_j := k'+1] &\Rightarrow \langle test(i,j,k), assign(j,k'), v\rangle \in D_{\tau_j} && (12)
\end{aligned}
$$

Equation (12) says that there is a dependency between a test and an assignment iff the assignment changes the result of the test. The conditional dependency relation for p_i is the smallest symmetric (in the first two arguments) ternary relation such that

$$a \neq b \quad \Rightarrow \quad \langle p_i := a, p_i := b, v\rangle \in D_{p_i} \tag{13}$$

$$a \neq v \quad \Rightarrow \quad \langle \mathit{read}(p_i), p_i := a, v\rangle \in D_{p_i} \tag{14}$$

Consider the relation $\triangleright_s$. The fact that the sequence of vector timestamps in a local computation is non-decreasing implies that, if a conjunct *op* of the form $c[j][\tau_j](vt)[j] \geq c[i][k+1](vt)[j]$ in the guard of a transition t is true in a state s, then a subsequent write operation op' to τ_j cannot falsify it, so $op \not\triangleright_s op'$. Thus, by exploiting enough information about the structure of the system, $\triangleright_s$ can be refined to avoid the above problem. To see this, consider applying CSSA in a state s. In step 1, we choose a transition t_e that is enabled in s, and take $T_s = \{t_e\}$. We execute step 2 for the first time, with $t = t_e$. Since t_e is enabled in s, we execute step 2(b). Assume the transitions have been re-defined as in Section 9.1.1. Step 2(b)i adds no transitions to T_s, because no transition $t' \neq t_e$ is in conflict with t_e. For each $j \notin \{0, i\}$, let t'_j be the next transition of process j, *i.e.*, $t'_j = t_{j,s(\tau_j)}$. Since t_e is enabled in s, the operations in its guard are true, so the write to τ_j in the command of t'_j does not interfere with those operations, so this first execution of step 2(b)ii does not necessarily add t'_j to T_s.

t_0 and t_e are parallel [God96, p. 45] and the read from p_i in the guard of t_0 might be the first operation to interfere with the write to p_i in the command of t_e from s, so step 2(b)ii adds t_0 to T_s. So, we execute step 2 again, this time with $t = t_0$. Suppose t_0 is disabled in s (otherwise, we can stop the search and report **Poss** Φ), so we execute step 2(a). Assuming the search is halted as soon as a state in which t_0 is enabled is encountered, it is easy to see that s contains $\ell_{0,nd}$, not $\ell_{0,d}$, so step 2(a)i is inapplicable, so we execute step 2(a)ii. In step 2(a)ii, we choose a conjunct ϕ in the guard of t_0 (*i.e.*, a conjunct of Φ) that is false in s, and for each i in $\mathit{supp}(\phi)$, we add to T_s the next transition of process i, namely, $t_{i,s(\tau_i)}$, since the write to p_i in the command of that transition might be the first to interfere with the read from p_i in t_0's guard. Let $t_{i,k}$ be one of the transitions so added to T_s. Consider executing step 2 with $t = t_{i,k}$. If $t_{i,k}$ is enabled in s, we execute step 2(b), which does not cause any new transitions to be added to T_s. Suppose $t_{i,k}$ is disabled in s, so we execute step 2(a). Since $t_{i,k}$ is the next transition of process i, we conclude that some conjunct *op* of the form $c[j][\tau_j](vt)[j] \geq c[i][k+1](vt)[j]$ in the guard of $t_{i,k}$ is false. Thus, step 2(a)i is inapplicable, and we choose the aforementioned conjunct *op* in step 2(a)ii. This execution of step 2(a)ii adds to T_s the first transition $t_{j,k'}$ of process j that truthifies *op*.

Suppose $t_{j,k'}$ appears near the end of the local computation of process j, and $s(\tau_j) < k'$. When executing step 2(a) with $t_{j,k'}$, we can choose step 2(a)i and add $t_{j,k'-1}$ to T_s, or, because the conjunct $\tau_j = k'$ is false, we can choose step 2(a)ii, which will have the same effect, namely, adding $t_{j,k'-1}$ to T_s, because the write to τ_j in the command of $t_{j,k'-1}$ is the first operation that might interfere with $\tau_j = k'$ from s. If the other conjuncts in the guard of $t_{j,k'}$ are true, then these are the only possible choices. When executing step 2 with $t_{j,k'-1}$, the same situation may arise. Eventually, this may add to T_s the transitions $t_{j,k'}, t_{j,k'-1}, t_{j,k'-2}, \ldots, t_{j,s(\tau_j)}$. Thus, the worst-case size of T_s is $\Theta(S)$, and the worst-case time complexity of CSSA is $\Theta(S)$, ignoring factors of N. In contrast, the worst-case time complexity of $\mathrm{PS}_{\mathbf{Poss}}$ is $O(Nd)$, independent of S.

9.2 Comparison to Other Algorithms

Since CSSA based on computation-specific operations is effective but expensive, it is natural to ask whether any of the simpler algorithms in [God96] would be equally effective and cheaper.

Stubborn Set Algorithm. The stubborn set algorithm [God96, Section 4.5] is ineffective, returning all enabled transitions, even if the transitions are re-defined as in Section 9.1.1 and the relations can-be-dependent and do-not-accord are based on the operations and dependency relations D_{τ_j} and D_{p_i} in Section 9.1.2. To see this, consider applying CSSA in a state s. In step 1, we choose a transition $t_{i,k}$ that is enabled in s, and take $T_s = \{t_{i,k}\}$. We execute step 2 for the first time, with $t = t_{i,k}$. Since $t_{i,k}$ is enabled in s, we execute step 2(b). Let $j \notin \{0, i\}$, and let $k' = s(\tau_j)$. Note that $t_{j,k'}$ is the next transition of process j, and that $t_{j,k'}$ uses $\mathit{assign}(j, k')$. We argue that $\mathit{test}(i, j, k)$ and $\mathit{assign}(j, k')$ do-not-accord. Suppose $\mathit{assign}(j, k')$ makes $\mathit{test}(i, j, k)$ true; then, in any state s in which $\mathit{test}(i, j, k)$ is false, $\mathit{assign}(j, k')$ and $\mathit{test}(i, j, k)$ are enabled and dependent. Suppose $\mathit{assign}(j, k')$ makes $\mathit{test}(i, j, k)$ false; then, in any state s in which $\mathit{test}(i, j, k)$ is true, $\mathit{assign}(j, k')$ and $\mathit{test}(i, j, k)$ are enabled and dependent. Thus, in either case, $\mathit{test}(i, j, k)$ and $\mathit{assign}(j, k')$ do-not-accord, so step 2(b)ii adds $t_{j,k'}$ to T_s. Thus, the algorithm returns $\mathit{enabled}(s)$.

Overman's Algorithm. Overman's Algorithm [God96, Section 4.4] is ineffective, returning all enabled transitions, even if the transitions are re-defined as in Section 9.1.1 and the relation $\triangleright_s$ of Section 9.1.2 is used instead of the less precise can-be-dependent relation (*cf.* [God96, Section 4.8]). Theorem 4.20 in [God96, Section 4.6] then implies that the Conflicting Transitions Algorithm [God96, Section 4.3] is also ineffective. Consider applying Overman's Algorithm in a state s. In step 1, we choose a transition $t_{i,k}$ that is enabled in s, and take $P = \{i\}$. Execute step 2 for the first time with $t = t_{i,k}$. Step 2(a) adds nothing to P. Consider step 2(b) with $t' = t_0$. $t_{i,k}$ and t_0 are parallel [God96, p. 45] and the read from p_i in the guard of t_0 might be the first operation to interfere with the write to p_i in the command of $t_{i,k}$ from s, so step 2(b) inserts 0 in P. Execute step 2 for the second time with $t = t_0$. Step 2(a) has no effect. For each process $j \notin \{0, i\}$, consider step 2(b) with t' being the next transition of process j, *i.e.*, $t' = t_{j,s(\tau_j)}$. t_0 and t' are parallel, and the write to p_j in the command of t' might be the first operation to interfere with the read from p_j in the guard of t_0 from s, so step 2(b) inserts j in P. Thus, Overman's Algorithm returns $\mathit{enabled}(s)$.

9.3 Persistent Sets for Def

We consider using CSSA instead of $\mathrm{PS}_{\mathbf{Def}}$ for detecting $\mathbf{Def}\,\Phi$, and conclude that CSSA is ineffective, returning all enabled transitions. The comparisons in [God96, Sections 4.6, 4.8] then imply that the simpler persistent-set algorithms in [God96] are also ineffective.

We adopt the same operations on shared variables as in Section 9.1.2, *i.e.*, computation-specific operations for $\tau_1, \ldots, \tau_N$, and read and write operations for $p_1, \ldots, p_N$. Consider applying CSSA in a state s. In step 1, we choose a transition t_e that is enabled in s, and take $T_s = \{t_e\}$. We execute step 2 for the first time, with $t = t_e$. Since t_e is enabled in s, we execute step 2(b). Step 2(b)i adds no transitions to T_s, because no transition $t' \neq t_e$ is in conflict with t_e. Let $t_{j,k}$ be a transition other than t_e that is enabled in s. The command of $t_{j,k}$ writes to p_j, and the command of t_e reads from p_j (in the assignment statement that updates h). The former operation might be the first to interfere with the latter operation from s, so step 2(b)ii adds $t_{j,k}$ to T_s. Thus, CSSA returns $\mathit{enabled}(s)$.

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.

[BR94] Özalp Babaoğlu and Michel Raynal. Specification and verification of behavioral patterns in distributed computations. In *Fourth International Working Conference on Dependable Computing for Critical Applications*, 1994.

[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.

[Hel99] Keijo Heljanko. Using logic programs with stable model semantics to solve deadlock and reachability problems for 1-safe petri nets. In W. Rance Cleaveland, editor, *Proc. 5th Intl. Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 1579 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.

[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.

[SL98] Scott D. Stoller and Yanhong A. Liu. Efficient symbolic detection of global properties in distributed systems. In Alan J. Hu and Moshe Y. Vardi, editors, *Proc. 10th Int'l. Conference on Computer-Aided Verification (CAV)*, volume 1427 of *Lecture Notes in Computer Science*, pages 357–368. Springer-Verlag, June 1998.

[SS95] Scott D. Stoller and Fred B. Schneider. Faster possibility detection by combining two approaches. In Jean-Michel Hélary and Michel Raynal, editors, *Proc. 9th International Workshop on Distributed Algorithms (WDAG '95)*, volume 972 of *Lecture Notes in Computer Science*, pages 318–332. Springer-Verlag, September 1995.

[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: We start with some observations. No enabled transition ever becomes disabled. After h becomes true, its value never changes. Two transitions that are both enabled in a state s are dependent in s only if the order they are executed affects the final value of h, *i.e.*, only if $s \xrightarrow{t_1} s_1 \xrightarrow{t_2} s_{12}$ and $s \xrightarrow{t_2} s_2 \xrightarrow{t_1} s_{21}$ and $s(h) =$ false and exactly one of s_1 and s_2 satisfies Φ.

Consider each **return** statement in $\mathrm{PS}_{\mathbf{Def}}$. For the **return** statements in lines (a) and (f), correctness follows from the fact that $\mathit{enabled}(s)$ is persistent in s. For (b), correctness follows easily from the above observations.

For (c), suppose $s_1 = s$ and $s_1 \xrightarrow{t_1} s_2 \xrightarrow{t_2} s_3 \cdots \xrightarrow{t_{n-1}} s_n \xrightarrow{t_n} s_{n+1}$ and $(\forall i \in [1..n] : t_i \notin T)$. Suppose $t_p \in T$ and $s_n \xrightarrow{t_p} s_p \xrightarrow{t_n} s_{pn}$ and $s_n \xrightarrow{t_n} s_{n+1} \xrightarrow{t_p} s_{np}$ (this defines s_n, s_{pn}, and s_{np}; note that these execution fragments are feasible, *i.e.*, t_n and t_p are enabled in the appropriate states). Based on the above observations, it suffices to show that $s_p(h) = s_{n+1}(h)$. In branches $(c) - -(e)$, $s(h) =$ false, and the definition of T implies that ϕ remains false for all sequences of transitions that stay outside of T, so $s_{n+1}(h) =$ false, so it suffices to show that $s_p \not\models \Phi$. The definition of T implies that $(\forall i \in \mathit{supp}(\phi) : s_n(\tau_i) = s(\tau_i))$, the condition on line (c) of the pseudo-code implies that executing t_p in s or s_n leaves ϕ false, so $s_p \not\models \phi$, so $s_p \not\models \Phi$.

For (d), consider the same setup as for (c). Again, it suffices to show that $s_p \not\models \Phi$. Since ϕ' is 1-local and $s \not\models \phi'$, $\mathit{falseUntilEnd}(\phi', s, T)$ ensures that ϕ' remains false for all sequences of transitions starting from s and staying outside T, except possibly in the last state of the sequence. To see this, note that the first conjunct of *falseUntilEnd* ensures that only the last transition of process j in such a sequence could truthify ϕ', and the second conjunct ensures that the sequence contains only transitions of process j, because every other process is either in $\mathrm{fl}(T)$ or waiting for a process in $\mathrm{fl}(T)$. Thus, s_{n+1} might satisfy ϕ', but $s_n \not\models \phi'$. If $\mathrm{fl}(t_p) \notin \mathit{supp}(\phi')$, then t_p does not affect ϕ', so $s_p \not\models \phi'$. If $\mathrm{fl}(t_p) \in \mathit{supp}(\phi')$, then $\mathrm{fl}(t_p) \notin \mathit{supp}(\phi)$ (because $\mathit{supp}(\phi')$ and $\mathit{supp}(\phi)$ are disjoint); the definition of T implies $s_n \not\models \phi$, so $s_p \not\models \phi$.

For (e), consider the same setup as for (c). Again, it suffices to show that $s_p \not\models \Phi$. By definition of $T \cup \mathrm{followWF}(s, \mathit{supp}(\phi'))$, $s_n \not\models \phi$ and $s_n \not\models \phi'$. $\mathit{supp}(\phi')$ and $\mathit{supp}(\phi)$ are disjoint, so a single transition can truthify at most one of ϕ and ϕ', so $s_p \not\models \Phi$. ∎