# Efficient Symbolic Detection of Global Properties in Distributed Systems

Scott D. Stoller* and Yanhong A. Liu
Computer Science Dept, Indiana University

6 February 1998

**Abstract**

A new approach is presented for detecting whether a computation of an asynchronous distributed system satisfies **Poss** $\Phi$ (read "possibly $\Phi$"), meaning the system could have passed through a global state satisfying property $\Phi$. Previous general-purpose algorithms for this problem explicitly enumerate the set of global states through which the system could have passed during the computation. The new approach is to represent this set symbolically, in particular, using ordered binary decision diagrams. We describe an implementation of this approach, suitable for off-line detection of properties, and compare its performance to the enumeration-based algorithm of Alagar & Venkatesan. In typical cases, the new algorithm is significantly faster. We have measured over 400-fold speedup in some cases.

## 1 Introduction

A history of a distributed system can be modeled as a sequence of events in their order of occurrence. Since execution of a particular sequence of events leaves the system in a well-defined global state, a history uniquely determines a sequence of global states through which the system has passed. Unfortunately, in an asynchronous distributed system,[1] no process can determine the order in which events on different processors actually occurred. Therefore, no process can determine the sequence of global states through which the system passed. This leads to an obvious difficulty for detecting whether a global property (*i.e.*, a predicate on global states) 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 $\rightarrow$, 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 [Mat89], 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 are exactly those sequences that correspond to total orders containing the happened-before relation. A *consistent global state* (CGS) of a computation $c$ is a global state that occurs in some history consistent with $c$. A computation satisfies **Poss** $\Phi$ iff, in *some* history consistent with that computation, the system passes through a global state satisfying $\Phi$. A computation satisfies **Def** $\Phi$ iff, in *all* histories consistent with that computation, the system passes through a global state satisfying $\Phi$.

Cooper and Marzullo give centralized algorithms for detecting **Poss** $\Phi$ and **Def** $\Phi$ for an arbitrary predicate $\Phi$ [CM91]. A stub at each process reports the local states of that process to a central monitor. The central monitor incrementally constructs a lattice whose elements correspond to CGSs of the computation. A straightforward search of the lattice reveals whether the computation satisfies **Poss** $\Phi$ or **Def** $\Phi$.

Unfortunately, these algorithms can be expensive. In a system of $N$ processes, the worst-case number of CGSs is $\Theta(S^N)$, where $S$ is the maximum number of steps taken by a single process. This worst case

*Address: Lindley Hall 215, Indiana University, Bloomington, IN 47405, USA. Email: stoller@cs.indiana.edu.

[1] An *asynchronous* distributed system is characterized by lack of synchronized clocks and lack of bounds on processor speed and network latency.

comes from the (exponential) number of CGSs of a computation in which there is little communication. Any detection algorithm that enumerates all CGSs—whether using the method in [CM91, MN91] or one of the more efficient schemes in [JMN95, AV97]—has time complexity that is at least linear in the number of CGSs. This time complexity can be prohibitive, so researchers have sought faster alternatives. One approach has been to restrict the problem and develop efficient algorithms for detecting only certain classes of predicates [GW94, TG93].[2] Another approach has been to modify some aspect of the problem—for example, detecting a different modality [FR94a] or assuming that the system is partially synchronous [MN91, Sto97].

This paper presents an efficient and general approach to detecting **Poss** $\Phi$. In this approach, the set of CGSs is represented symbolically, using ordered binary decision diagrams. This can be much more efficient than explicit enumeration. For simplicity, we consider here only off-line detection, in which the detection algorithm is run after the distributed computation has terminated. The approach can also be applied to on-line detection. Section 2 provides some background. Section 3 describes our basic and optimized detection algorithms. Section 4 gives performance results from using the new algorithm and an enumeration-based algorithm [AV97] to detect violations of invariants in a coherence protocol and a spanning-tree algorithm. For both examples, when the invariant is not violated, the new method is faster by a factor that increases exponentially with the number of processes in the system. We also measure the effects of judiciously applying the two variable-reordering methods supplied by the BDD library. Both methods greatly reduce memory consumption, though at a significant cost in running time. Performance measurements also show that our optimized detection algorithm is much faster than our basic algorithm for one example but is about the same speed as the basic algorithm for the other example. Directions for future work include characterizing the class of examples for which the optimization is effective, extending the algorithms to support on-line detection, applying the approach to detection of **Def** $\Phi$, and investigating more optimizations.

# 2 Background

## 2.1 System Model

A (distributed) system is a collection of processes connected by an asynchronous, reliable, and FIFO network. Let $N$ denote the number of processes. We use the numbers $0, 1, \ldots, N-1$ as process identifiers, and define $\mathrm{PID} = \{0, 1, \ldots, N-1\}$. A local state $s$ of a process $p$ is a mapping from the local variables of $p$ to values; for example, $s(x)$ is the value of variable $x$ in local state $s$.

Each process starts in a specified initial state and optionally with its timer set to a specified value. Computations contain only two kinds of events: timer expiration and message reception. As a result of either kind of event, a process can atomically (*i.e.*, without interruption by other events) change its local state, send a set of messages (with specified destinations), and set its timer.[3] Processes can be non-deterministic, *i.e.*, the input event need not uniquely determine the new local state, set of sent messages, and timer setting.

Each process has a timer. For convenience, we assume the timers all run at the same speed, though this assumption is not required for correctness of the example protocols in Section 4.

Each process $p$ has a vector clock $vc_p$ with $N$ components. We regard $vc_p$ as a (special) variable; thus, $s(vc_p)$ is the value of the vector clock in local state $s$. In the initial state of process $p$, $vc_p = \langle 0, 0, \ldots, 0 \rangle$. The vector clock is updated after each event, and the updated value is piggybacked on the outgoing messages (if any). Thus, each message $m$ has a vector timestamp $\mathrm{ts}(m)$. The rules for updating the vector clock are: (1) For a timerExpire event of process $p$, component $p$ of $vc$ is incremented by 1; (2) When process $p$ receives a message $m$, its vector clock $vc_p$ is assigned the component-wise maximum $\max(vc_p, \mathrm{ts}(m))$ and then component $p$ is incremented by 1.

Given a system, a straightforward simulation can be used to generate a possible computation of that system. The intrinsic non-determinism of the asynchronous network is modeled by selecting message latencies from a random distribution. Each running timer and in-transit message corresponds to a pending event. When a pending event is generated, it is timestamped with its (future) time of occurrence. The simulator

[2] These restricted algorithms do not apply to the predicates detected in the examples in Section 4.

[3] Thus, in contrast to most models of distributed computation, the sending of a message is not modeled as a separate event. This difference is inessential but simplifies our model slightly.

repeatedly executes the pending event with the lowest timestamp, thereby changing the local state of a process and generating new pending events. Since some protocols are designed to service requests forever, the simulator accepts a parameter *maxlen*, which is the maximum number of events per process. So, the simulation ends either when there are no pending events or when some process has executed *maxlen* events.

A *computation* of a system is represented as a sequence of $N$ local computations, one per process. A *local computation* is a sequence of local states that represents the execution history of a single process. Each local state includes values of all the declared variables of the process and (implicitly) the value of the process's vector clock (*i.e.*, the value of the vector clock after updating by the input event that led to this local state).

### 2.2 Consistent Global States and Poss $\Phi$

A *global state* is a collection of local states, one from each process. For a sequence $c$ and natural number $i$, $c[i]$ denotes the $i$'th element of $c$ (we use 0-based indexing for sequences). A global state of a computation $c$ is a collection of local states $s_0, \ldots, s_{N-1}$ such that, for each process $p$, $s_p$ is an element of $c[p]$.

Some global states of a computation are uninteresting, because the system could not have been in those global states during that computation. So, we restrict attention to *consistent* global states, *i.e.*, global states through which the system might have passed during the computation. We define consistency for global states in terms of the happened-before relation on local states [GW94]. Intuitively, a local state $s_1$ happened-before a local state $s_2$ (of the same or a different process) if $s_1$ finished before $s_2$ started. In particular, define $\rightarrow$ for a computation $c$ to be the smallest transitive relation on the local states of $c$ such that

1. For all proceses $p$ and all local states $s_1$ and $s_2$ of $p$ in $c$, if $s_1$ immediately precedes $s_2$, then $s_1 \rightarrow s_2$.
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 $s_1 \rightarrow s_2$.

Two local states of a computation are *concurrent* iff neither "happened before" the other: $s_1 \parallel s_2 \triangleq s_1 \not\rightarrow s_2 \;\wedge\; s_2 \not\rightarrow s_1$. A global state is *consistent* iff its constituent local states are pairwise concurrent.

Vector timestamps are useful because they capture the happened-before relation [Mat89]. Define a partial order $\prec$ on vector timestamps by: $v_1 \prec v_2$ iff $(\forall p \in \mathrm{PID} : v_1[p] \leq v_2[p])$. Then, for all computations $c$ and all processes $p_1$ and $p_2$,

$$(\forall i_1 \in \mathrm{dom}(c[p_1]), i_2 \in \mathrm{dom}(c[p_2]) : c[p_1][i_1] \rightarrow c[p_2][i_2] \;\;\equiv\;\; c[p_1][i_1](vc_{p_1}) \prec c[p_2][i_2](vc_{p_2})) \qquad (1)$$

where for a sequence $\sigma$, $\mathrm{dom}(\sigma)$ is the domain of $\sigma$, *i.e.*, $\mathrm{dom}(\sigma) = \{0, 1, \ldots, (|\sigma| - 1)\}$, where $|\sigma|$ is the length of $\sigma$. Thus, concurrency of two local states can be tested in $O(N)$ time using vector timestamps. Concurrency of two local states can be tested in constant time by exploiting the following theorem [FR94b]: for a local state $s_1$ of process $p_1$ and a local state $s_2$ of process $p_2$,

$$s_1 \parallel s_2 \;\;\equiv\;\; s_1(vc_{p_1})[p_2] \leq s_2(vc_{p_2})[p_2] \;\wedge\; s_2(vc_{p_2})[p_1] \leq s_1(vc_{p_1})[p_1] \qquad (2)$$

where (for example) $s_1(vc_{p_1})[p_2]$ is component $p_2$ of the vector timestamp $s_1(vc_{p_1})$.

The definition of **Poss** is: a computation $c$ satisfies **Poss** $\Phi$ iff there exists a consistent global state of $c$ that satisfies $\Phi$. We write $c \models$ **Poss** $\Phi$ to denote that computation $c$ satisfies **Poss** $\Phi$.

## 3 Detection Method

To detect $c \models$ **Poss** $\Phi$ efficiently using symbolic methods, we generate a formula $b$ such that $b$ is satisfiable iff $c \models$ **Poss** $\Phi$. In this formula, we use $x_p$ to denote the local variables (excluding the vector clock) of process $p$, and we use the variable $vc_{p,q}$ to denote component $q$ of the vector clock of process $p$ (*i.e.*, we treat each vector clock as $N$ separate variables). For convenience, we assume that the sets of local variables of different processes are disjoint. Let $\vec{x}$ denote the collection of variables $x_0, x_1, \ldots, x_{N-1}$, and let $\vec{vc}$ denote the collection of all $\Theta(N^2)$ vector-clock variables. Using (2) to express concurrency of local states, it is easy to show that $b$ can be taken to be

$$\Phi(\vec{x}) \;\wedge\; \mathrm{globalState}_c(\vec{x}, \vec{vc}) \;\wedge\; \mathrm{consis}_c(\vec{vc}) \qquad (3)$$

where

$$\begin{aligned} \text{globalState}_c(\vec{x}, \vec{vc}) &= \bigwedge_{p \in \text{PID}} \bigvee_{i \in \text{dom}(c[p])} x_p = c[p][i](x_p) \ \wedge\ (\bigwedge_{q \in \text{PID}} vc_{p,q} = c[p][i](vc_p)[q]) && (4) \\ \text{consis}_c(\vec{vc}) &= \bigwedge_{p_1 \in \text{PID}} \bigwedge_{p_2 \in (\text{PID} \setminus \{p_1\})} vc_{p_2,p_1} \leq vc_{p_1,p_1} && (5) \end{aligned}$$

Since we test $c \models \mathbf{Poss}\,\Phi$ by testing satisfiability of $b$, there is, in effect, an implicit existential quantification over all of the free variables of $b$.

For example, consider a system with $N = 2$. Suppose each process $p$ has a single local variable $y_p$, and that we want to detect $\mathbf{Poss}(x_0 + x_1 = 1)$. Consider the computation $c$ in which each local computation has length 2, and $c[p][i](y_p) = i$, $c[p][0](vc_p) = \langle 0, 0\rangle$, $c[0][1](vc_0) = \langle 1, 0\rangle$, and $c[1][1](vc_1) = \langle 1, 1\rangle$. Instantiating (3), we obtain the formula

$$(y_0 + y_1 = 1) \ \wedge\ \text{globalState}_c(y_0, y_1, \vec{vc}) \ \wedge\ \text{consis}_c(\vec{vc}) \tag{6}$$

where

$$\begin{aligned} \text{globalState}_c(y_0, y_1, \vec{vc}) &= \quad ((y_0 = 0 \ \wedge\ (vc_{0,0} = 0 \ \wedge\ vc_{0,1} = 0)) \vee (y_0 = 1 \ \wedge\ (vc_{0,0} = 1 \ \wedge\ vc_{0,1} = 0))) \\ &\quad \wedge\ ((y_1 = 0 \ \wedge\ (vc_{1,0} = 0 \ \wedge\ vc_{1,1} = 0)) \vee (y_1 = 1 \ \wedge\ (vc_{1,0} = 1 \ \wedge\ vc_{1,1} = 1))) \\ \text{consis}_c(\vec{vc}) &= \ vc_{0,1} \leq vc_{1,1} \ \wedge\ vc_{1,0} \leq vc_{0,0} \end{aligned}$$

Formulas obtained from (3) contain $\Theta(N^2)$ variables for the vector clocks. To reduce the number of variables in the formula, and thereby reduce the cost of testing satisfiability of the formula, we use a change of variables. For each process $p$, we introduce a new variable $idx_p$, which contains the "index" of the local state in $c[p]$, *i.e.*, $(\forall i \in \text{dom}(c[p]) : c[p][i](idx_p) = i)$.[4] Re-expressing globalState and consis in terms of these new variables, we take $b$ to be:

$$\Phi(\vec{x}) \ \wedge\ \text{globalState}_c(\vec{x}, \vec{idx}) \ \wedge\ \text{consis}_c(\vec{idx}) \tag{7}$$

where

$$\begin{aligned} \text{globalState}_c(\vec{x}, \vec{idx}) &= \bigwedge_{p \in \text{PID}} \bigvee_{i \in \text{dom}(c[p])} x_p = c[p][i](x_p) \ \wedge\ idx_p = i && (8) \\ \text{consis}_c(\vec{idx}) &= \bigwedge_{p_1 \in \text{PID}} \bigwedge_{p_2 \in (\text{PID} \setminus \{p_1\})} \bigvee_{i_2 \in \text{dom}(c[p_2])} idx_{p_2} = i_2 \ \wedge\ c[p_2][i_2](vc_{p_2})[p_1] \leq idx_{p_1} && (9) \end{aligned}$$

where $\vec{idx}$ denotes the collection of variables $idx_0, idx_1, \ldots, idx_{N-1}$.

Revisiting the above example, we obtain, instead of (6), the formula

$$(y_0 + y_1 = 1) \ \wedge\ \text{globalState}_c(y_0, y_1, \vec{idx}) \ \wedge\ \text{consis}_c(\vec{idx}) \tag{10}$$

where

$$\begin{aligned} \text{globalState}_c(y_0, y_1, \vec{idx}) &= \quad ((y_0 = 0 \ \wedge\ idx_0 = 0) \vee (y_0 = 1 \ \wedge\ idx_0 = 1)) \\ &\quad \wedge ((y_1 = 0 \ \wedge\ idx_1 = 0) \vee (y_1 = 1 \ \wedge\ idx_1 = 1)) \\ \text{consis}_c(\vec{idx}) &= \quad ((idx_1 = 0 \ \wedge\ 0 \leq idx_0) \vee (idx_1 = 1 \ \wedge\ 1 \leq idx_0)) \\ &\quad \wedge ((idx_0 = 0 \ \wedge\ 0 \leq idx_1) \vee (idx_0 = 1 \ \wedge\ 0 \leq idx_1)) \end{aligned}$$

## 3.1 Implementation and an Optimization

We represent the formula defined by (7) using ordered binary decision diagrams (BDDs) [Bry92]. The main benefits of this representation are: (1) it is a canonical form, so testing satisfiability is easy; (2) for

[4] We could take $idx_p$ to be $vc_{p,p}$, since the rules for updating vector clocks imply that $c[p][i](vc_{p,p}) = i$. However, we find it more straightforward to think of $idx_p$ as a new variable.

many formulas of interest, BDDs are more compact than other canonical forms (such as conjunctive normal form).

Let $\mathrm{true}_{\mathrm{bdd}}$ and $\mathrm{false}_{\mathrm{bdd}}$ denote the BDDs representing true and false, respectively. Let $\wedge_{\mathrm{bdd}}$ denote conjunction of BDDs. Let a formula with an overline denote a function that returns the BDD representation of that formula. Formula $b$ can be constructed and tested for satisfiability by the following pseudo-code:

```
procedure BDD-detection0($c, \overline{\Phi}$)
    $b := \mathrm{true}_{\mathrm{bdd}}$
    $b := b \wedge_{\mathrm{bdd}} \overline{\mathrm{globalState}_c}(\vec{x}, \vec{idx})$
    $b := b \wedge_{\mathrm{bdd}} \overline{\mathrm{consis}_c}(\vec{idx})$
    $b := b \wedge_{\mathrm{bdd}} \overline{\Phi}(\vec{x})$
    if $b = \mathrm{false}_{\mathrm{bdd}}$ then return("$c \not\models \mathbf{Poss}(\Phi)$") else return("$c \models \mathbf{Poss}(\Phi)$")
```

The functions $\overline{\mathrm{globalState}}$ and $\overline{\mathrm{consis}}$ are easily written based on (8) and (9). The numbers in vector timestamps are encoded as unsigned integers, with a binary variable representing each bit; the number of bits required is easily determined, since we consider here only off-line detection. Currently, the user writes the function $\overline{\Phi}$, though generating that function automatically from a logical formula would be straightforward. If $\mathbf{Poss}\,\Phi$ holds, it is straightforward to obtain a satisfying assignment for $b$ and (from that) a particular CGS satisfying $\Phi$.

We implemented also an optimized version of the above procedure. Often (as in both examples in Section 4), $\Phi$ is a disjunction: $\Phi = \bigvee_{\alpha \in S} \Phi_\alpha$, where $S$ is a set. Since conjunction distributes over disjunction, we can re-write the detection procedure as

```
procedure BDD-detection($c, \bigvee_{\alpha \in S} \overline{\Phi}_\alpha$)
    $b := \mathrm{true}_{\mathrm{bdd}}$
    $b := b \wedge_{\mathrm{bdd}} \overline{\mathrm{globalState}_c}(\vec{x}, \vec{idx})$
    $b := b \wedge_{\mathrm{bdd}} \overline{\mathrm{consis}_c}(\vec{idx})$
    for each $\alpha$ in $S$
        $b1 := b \wedge_{\mathrm{bdd}} \overline{\Phi}_\alpha(\vec{x})$
        if $b1 \neq \mathrm{false}_{\mathrm{bdd}}$ then return("$c \models \mathbf{Poss}(\Phi)$")
    return("$c \not\models \mathbf{Poss}(\Phi)$")
```

By testing each disjunct of $\Phi$ separately, BDD-detection avoids constructing the potentially large intermediate result $\overline{\Phi}$.

# 4 Examples

We compare the performance of procedure BDD-detection to Alagar & Venkatesan's off-line detection algorithm [AV97], which (to our knowledge) is the most time- and space-efficient previously-known general-purpose algorithm for detecting **Poss**. Their algorithm, which we refer to as DFS-detection, performs an depth-first-search search of the lattice of consistent global states. Normally, a depth-first search requires storing all of the generated nodes (*i.e.*, consistent global states), in order to avoid re-exploring a node. The order in which their algorithm visits the successors (children) of a node is based on the vector timestamps, so one can determine from the vector timestamps whether a node has already been explored, thereby avoiding the need to store the set of explored nodes.

To characterize the performance of a detection algorithm, it is important to use a workload that includes both computations satisfying the property and computations not satisfying the property. The most common use of detection algorithms for **Poss** is to check that an invariant $I$ holds, by detecting whether the computation satisfies $\mathbf{Poss}\,\neg I$. To obtain computations in which the invariant is violated, we consider a buggy version of each example protocol (as well as the correct version).

For each version of each example, we use a simulator to generate a computation and then analyze that computation using both BDD-detection and DFS-detection. As mentioned above, the simulator selects message delays from a random distribution. We used the distribution $\rho_1 = 1 + \mathrm{expRand}(1)$, where $\mathrm{expRand}(\mu)$

generates random numbers from an exponential distribution with mean $\mu$. To measure the sensitivity of the analysis cost to message latencies, we considered also another (less realistic) distribution, $\rho_0 = \text{expRand}(1)$.

All measurements were made on a SGI Power Challenge with ten 75 MHz MIPS R8000 CPUs and 2GB RAM. The detection algorithms we measured are sequential, so the use of a parallel machine was irrelevant. We use the BDD library developed by E. Clarke's group at CMU [BDD]. The reported running times are the "user times" obtained from the UNIX `time` command; thus, they reflect the amount of CPU time consumed.

For BDD-detection, the variable ordering can affect performance. The overall variable ordering is $x_0, x_1, \ldots, x_N, idx_0, idx_1, \ldots, idx_N$, where $x_p$ denotes the sequence of binary variables encoding the local state of process $p$ excluding $idx_p$ and variables that are not mentioned in the predicate being detected, and $idx_p$ denotes the sequence of binary variables encoding the "index" of the local state.

## 4.1 Coherence Protocol

We consider a protocol that uses read locks and write locks to provide coherent access to shared data. The protocol allows concurrent reading of shared data, and it prevents a process from reading or writing shared data while another process is writing. Each process repeatedly tries to read or write the implicit shared data. Before starting to write, a process sends WriteReq to all other processes and waits for them to reply with WriteOK. On receiving WriteReq, a process replies immediately with WriteOK unless it is reading or writing or is waiting to write and had started waiting "before" the WriteReq was sent (as indicated by the relevant vector timestamps, compared using lexicographic order). If a process doesn't reply immediately to a WriteReq, it remembers the request and replies later. Before starting to read, a process waits for all processes to which it has sent WriteOK to reply with WriteDone. When a process starts reading or writing, it sets its timer to a value generated by expRand(4).[5] When the timer expires, the process stops reading or writing, respectively, and again sets its timer to a value generated by expRand(4). When the timer expires, the process tries to read or write (the choice is random) the shared data.

The buggy version of the protocol is the same except that WriteOK is included with every WriteDone. Each message is represented as a set of values—specifically, as a subset of {WriteReq, WriteOK, WriteDone}—so it is easy to modify the protocol in this way. To see why this modification is incorrect, consider a system with $N = 2$. Suppose process 0 is writing, and process 1 tries to start writing, so it sends WriteReq to everyone else. Before process 0 receives the WriteReq from process 1, process 0 finishes writing and sends {WriteDone,WriteOK} to process 1. Since process 0 still hasn't received WriteReq, it regards the WriteOK it just sent as a "no-op", so it can start reading immediately when its timer goes off. When process 1 receives the WriteOK from process 0, process 1 can immediately start writing. Thus, concurrent reading and writing can occur.

## 4.2 Analysis of Coherence Protocol

We use the detection algorithms to find violations of this invariant: when one process is writing, no other process is reading or writing. Thus, we detect the predicate

$$\Phi_C = \bigvee_{p_1 \in \text{PID}} \bigvee_{p_2 \in \text{PID} \setminus \{p_1\}} wrtg_{p_1} \wedge (rdg_{p_2} \vee wrtg_{p_2}).$$

where boolean variables $rdg_p$ and $wrtg_p$ indicate whether process $p$ is reading or writing, respectively.

To make the computations of the coherence protocol finite, we take the argument *maxlen* of the simulator to be $8N$; on average, this lets each process read or write the shared data twice during a computation. The running times of the detection algorithms, for computations with latency distribution $\rho_1$, are shown in Figure 1. For each detection algorithm, the corresponding curve and error bars show the average running time and the standard deviation, respectively, for 10 different seeds of the random number generator.

The graph on the left shows running times in seconds. Let $t_{\text{BDD}}(N)$ and $t_{\text{DFS}}(N)$ denote the running times of BDD-detection and DFS-detection, respectively. Both of these functions exhibit exponential growth—not surprising, since the number of consistent global states is exponential in $N$. Nevertheless, for larger values

[5] The choice of this distribution is arbitrary, in the sense that correctness of the protocol does not depend on it.

of $N$, the difference in the running times of the two procedures is dramatic. For example $t_{\mathrm{DFS}}(9)/t_{\mathrm{BDD}}(9) = (61268.3 \text{ sec}/141.6 \text{ sec}) \approx 433$; that is, the BDD algorithm is 433 times faster, running in about 2.4 minutes, compared to 17 hours. More generally, the ratio $t_{\mathrm{DFS}}(N)/t_{\mathrm{BDD}}(N)$ increases exponentially with $N$. This can be seen from the graph on the right, which shows the logarithm of the running times of both algorithms as a function of $N$. The curves for both algorithms are (roughly) lines; the smaller slope of the line for the BDD algorithm shows that its running time grows with a smaller exponent. This exponential growth in the ratio of running times occurs also when the latency distribution $\rho_0$ is used.

Now consider the buggy coherence protocol, with latency distribution $\rho_1$. We consider here only computations in which the bug manifests itself in a violation of the above invariant. BDD-detection is again faster than DFS-detection, though by a smaller margin; for example, it is faster by a factor of 46 at $N = 9$. The running time of BDD-detection is roughly independent of whether $c \models \textbf{Poss}\,\Phi_C$ holds. In contrast, the average running time of DFS-detection is significantly reduced (*e.g.*, by a factor of 7 to 10 for the buggy coherence protocol) when $c \models \textbf{Poss}\,\Phi_C$ holds, because DFS-detection halts as soon as it finds a consistent global state satisfying the predicate, and with luck, that can happen early in the search.

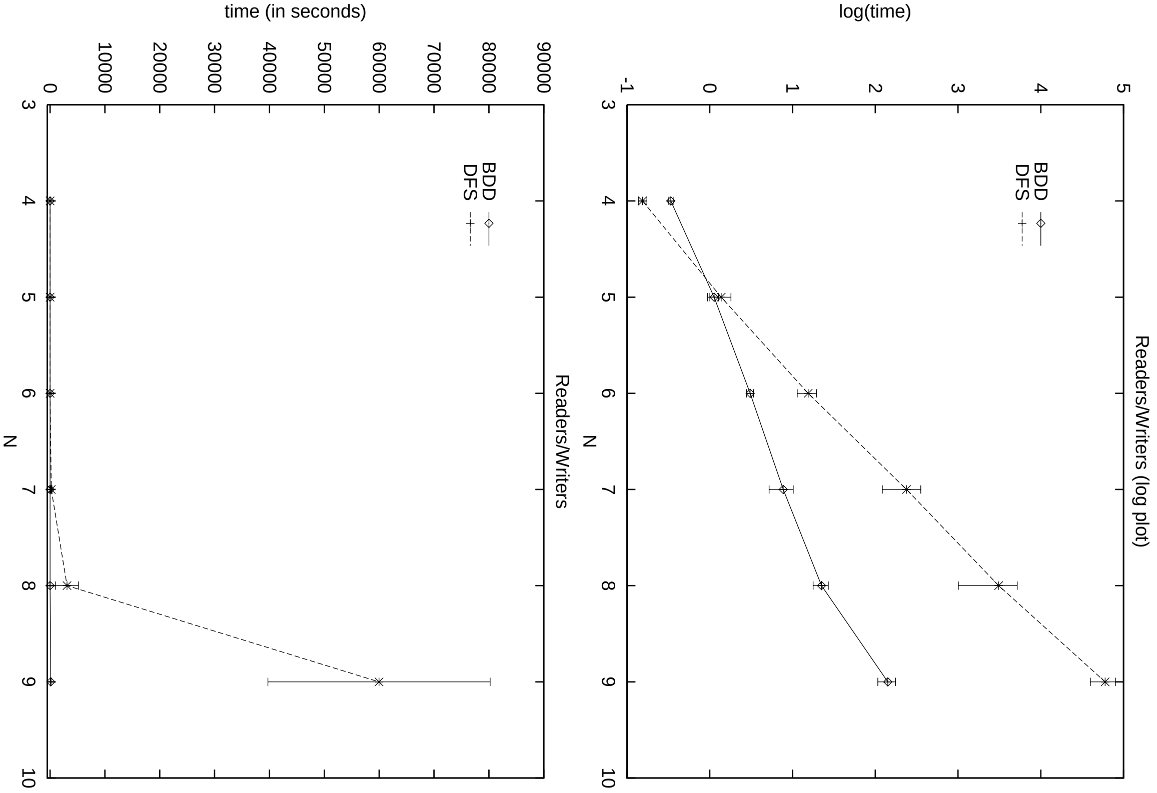

Figure 1: Coherence-protocol example. Left: Running time of detection algorithms. Right: Logarithm of running time of detection algorithms.

## 4.3 Spanning Tree

The following algorithm constructs a spanning tree in a network [Lyn96, Section 15.3]. For convenience, we assume that process 0 always initiates the algorithm and therefore always becomes the root of the spanning tree. Process 0 initiates the algorithm by sending its level in the tree (namely, 0) to each of its neighbors in the network. When a process other than process 0 receives its first message, it takes the sender of that message as its parent, and sets its level to one plus the level of its parent, and sends its level to each of its neighbors, except its parent. A process ignores subsequent messages.

To save space in local states, we represent the identity of the parent using relative coordinates rather than absolute coordinates. For example, in a (2-dimensional) grid with $N$ processes, we can represent the parent with 2 bits (0=left neighbor, 1=upper neighbor, *etc.*), compared to $\log_2 N$ bits to store a PID. The type nborid corresponds to these relative coordinates. For a process $p$ and relative coordinate $r$, $\mathrm{PIDofNbor}(p, r)$ is the PID of the process with relative coordinate $r$ with respect to process $p$. If process $q$ is a neighbor of process $p$, then $\mathrm{nborOfPID}(p, q)$ is the relative coordinate of $q$ with respect to $p$. Thus, $\mathrm{PIDofNbor}(p, \mathrm{nborOfPID}(p, q)) = q$.

In the buggy version of the algorithm, process 0 "forgets" to retain its special role, so it accepts the sender of the first message it receives (if any) as its parent. If the initial message from process 0 to a neighbor $p$ has a high latency, then $p$ might receive a message from some other process $p_1$ before $p$ receives a message from process 0. In that case, process $p$ sends a message to process 0, and (because of the bug) process 0 takes process $p$ as its parent, creating a cycle. To make this error manifest itself more often, when simulating the spanning tree algorithm, we always take the latency of messages from process 0 to process 1 to be 5.

## 4.4 Analysis of Spanning Tree

We use the detection algorithms to find violations of the following invariant: the level of a process is larger than the level of its parent. This invariant implies absence of cycles. Thus, we detect the predicate

$$\Phi_S = \bigvee_{p_1 \in \mathrm{PID}} \mathit{hasParent}_{p_1} \ \wedge \ \mathit{level}_{p_1} \leq \mathit{level}_{\mathrm{PIDofNbor}(\mathit{parent}_{p_1})}$$

where boolean variable $\mathit{hasParent}_p$ indicates whether process $p$ has gotten a parent, $\mathit{parent}_p$ is the (relative coordinate of) the parent of process $p$, and $\mathit{level}_p$ is the level of process $p$ in the spanning tree.

This predicate cannot be expressed directly as a boolean formula using the given set of variables, because $\mathit{level}_{\mathrm{PIDofNbor}(\mathit{parent}_{p_1})}$ is not a particular variable. So, we use DFS-detection to detect $\Phi_S$ but use BDD-detection to detect the following logically equivalent predicate:

$$\Phi'_S = \bigvee_{p_1 \in \mathrm{PID}} \ \bigvee_{p_2 \in \mathrm{PID} \setminus \{p_1\}} \mathit{hasParent}_{p_1} \ \wedge \ \mathit{parent}_{p_1} = p_2 \ \wedge \ \mathit{level}_{p_1} \leq \mathit{level}_{p_2}.$$

We analyze computations of this algorithm in a network with a grid topology. Let $m = \lfloor \sqrt{N} \rfloor$. Each process is connected to its neighbors in the grid:

$$\begin{array}{llll}
0 & 1 & \ldots & m-1 \\
m & m+1 & \ldots & 2m-1 \\
\vdots & \vdots & \vdots & \vdots \\
\lfloor N/m \rfloor m & \ldots & N-1 &
\end{array}$$

The running times of the detection algorithms, for computations with latency distribution $\rho_1$, are shown in Figure 2. The curves and error bars again show the average running time and the standard deviation, respectively, for 10 different seeds of the random number generator. One can see from the graph on the left that BDD-detection is significantly faster for larger values of $N$; for example, $t_{\mathrm{DFS}}(20)/t_{\mathrm{BDD}}(20) = (28797.6\ \mathrm{sec}/1360.2\ \mathrm{sec}) \approx 21.2$. The ratio $t_{\mathrm{DFS}}(N)/t_{\mathrm{BDD}}(N)$ again increases exponentially with $N$, as can be seen from the graph of the logarithm of the running times on the right of Figure 2. This exponential growth in the ratio of running times occurs also when the latency distribution $\rho_0$ is used.

Now consider the buggy spanning-tree algorithm, with latency distribution $\rho_1$. As above, we consider only computations in which the bug actually manifests itself in a violation of the invariant. The running time of the BDD algorithm is again roughly independent of whether $c \models \textbf{Poss}\,\Phi$ holds (for $4 \leq N \leq 20$, the running time for seeds that cause $\textbf{Poss}\,\Phi'_S$ to hold is within a factor of 2 of the running time for other seeds). In contrast, DFS-detection gets "lucky" on this example and finds a global state satisfying $\Phi$ very early in the search: for $4 \leq N \leq 20$, the running time for seeds that cause $\textbf{Poss}\,\Phi_S$ to hold is at most 0.05

sec, which is about $10^5$ times faster than for other seeds. So, for this example, DFS-detection is much faster than BDD-detection when $\mathbf{Poss}\,\Phi_S$ holds and is much slower when $\mathbf{Poss}\,\Phi_S$ does not hold.

To investigate the effect of using relative coordinates instead of a PID to indicate a process's parent, we implemented the spanning-tree protocol both ways. Naturally, the effect on the running time of DFS-detection is negligible. With BDD-detection, the number of bits in the global state space, hence the number of variables in the BDD, is larger when PIDs are used; for example, for $N = 20$, the number of variables increases from about 200 to 240, *i.e.*, increases by about 20%. This is reflected in a typically 20% increase in both memory usage and running time of BDD-detection.

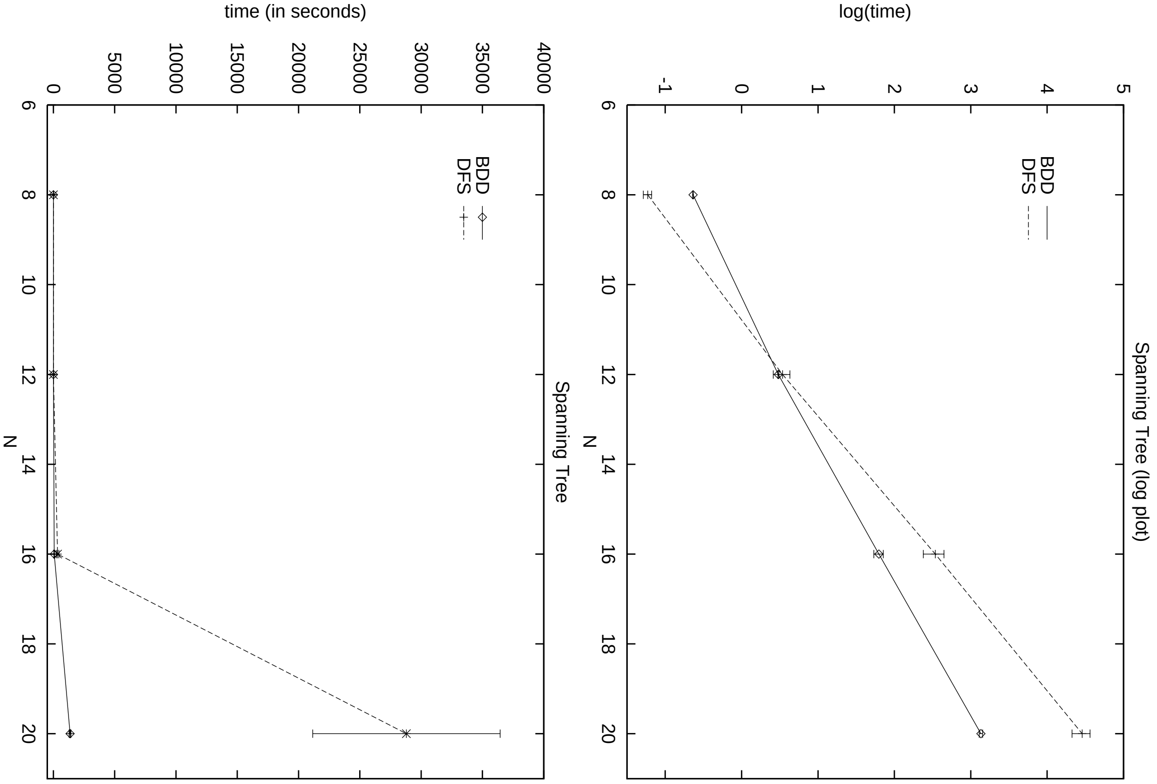

Figure 2: Spanning-tree example. Left: Running time of detection algorithms. Right: Logarithm of running time of detection algorithms.

## 4.5 Memory Usage

BDD-detection takes significantly more memory than DFS-detection, because BDD-detection constructs a BDD corresponding to the whole set of consistent global states, while DFS-detection requires only $O(N^2 S)$ space. Let $m_{\mathrm{BDD}}(N)$ and $m_{\mathrm{DFS}}(N)$ denote the memory used by BDD-detection and DFS-detection, respectively.

For the coherence protocol, $m_{\mathrm{BDD}}(N)$ grows exponentially with $N$, to 28.5 MB at $N = 9$, while $m_{\mathrm{DFS}}(N)$ is linear in $N$, growing to 2.6MB at $N = 9$. For the spanning-tree example, the same asymptotic behavior in memory usage occurs, though $m_{\mathrm{BDD}}(N)$ is much larger in absolute terms. For example, $m_{\mathrm{BDD}}(20) = 914MB$, while $m_{\mathrm{DFS}}(20) = 2.5MB$. The memory usage of BDD-detection can be greatly reduced by variable reordering, as described in Section 4.6.

## 4.6 Effect of Variable Reordering

We also ran BDD-detection using the two variable-reordering methods, called sift and window3, provided by the BDD package [BDD]. After constructing the BDD $\overline{\mathrm{globalState}_c}(\vec{x}, \vec{idx}) \wedge \overline{\mathrm{consis}_c}(\vec{idx})$, a variable-reordering method was applied to reduce the size of this BDD. (This is the only point at which variables were reordered.) According to the manual, the sift method "generally achieves greater size reductions, but is slower" (than window3). Let $t^{\mathrm{sift}}_{\mathrm{BDD}}(N)$ and $t^{\mathrm{win3}}_{\mathrm{BDD}}(N)$ denote the running times of BDD-detection with sift and window3 reordering, respectively, and let $m^{\mathrm{sift}}_{\mathrm{BDD}}(N)$ and $m^{\mathrm{win3}}_{\mathrm{BDD}}(N)$ denote the corresponding memory usage.

For the coherence protocol (with latency distribution $\rho_1$), $t^{\mathrm{sift}}_{\mathrm{BDD}}(N)/t_{\mathrm{BDD}}(N)$ is typically about 4, while $m^{\mathrm{sift}}_{\mathrm{BDD}}(N)/m_{\mathrm{BDD}}(N)$ is typically in the range 0.3–0.5. For example, for $N = 9$, the running time increased from 142 sec to 511 sec, and memory usage decreased from 29 MB to 9.0 MB. The window method caused a smaller increase in running time (as expected), with $t^{\mathrm{win3}}_{\mathrm{BDD}}(N)/tbdd(N)$ typically about 1.5, and, unexpectedly, achieved a greater decrease in memory usage, with $m^{\mathrm{win3}}_{\mathrm{BDD}}(N)/m_{\mathrm{BDD}}(N)$ typically in the range 0.2–0.4. For example, for $N = 9$, the running time increased from 142 sec to 203 sec, and memory usage decreased from 29 MB to 6.1 MB. So, for this example, the window method is preferable.

For the spanning-tree example (with latency distribution $\rho_1$), $t^{\mathrm{sift}}_{\mathrm{BDD}}(N)/tbdd(N)$ is typically about 6. Surprisingly, the window method caused a somewhat larger increase in running time, with $t^{\mathrm{win3}}_{\mathrm{BDD}}(N)/tbdd(N)$ typically about 9. Both reordering methods yield enormous reductions in memory usage, with greater fractional reductions at larger values of $N$. For example, $m^{\mathrm{sift}}_{\mathrm{BDD}}(9)/m_{\mathrm{BDD}}(9) = (914 \text{ MB}/48 \text{ MB}) \approx .05$, and $m^{\mathrm{sift}}_{\mathrm{BDD}}(9)/m_{\mathrm{BDD}}(9) = (914 \text{ MB}/69 \text{ MB}) \approx .08$. So, for this example, the sift method is preferable.

## 4.7 Comparing Performance of BDD-detection and BDD-detection0

Predicate $\Phi_C$ is a disjunction, so it is natural to expect that procedure BDD-detection offers a benefit over procedure BDD-detection0. However, when analyzing the correct coherence protocol (with latency distribution $\rho_1$), the two procedures have the same the running time and same amount of memory used, to within 1%. Similarly, for computations of the buggy coherence protocol that satisfy **Poss** $\Phi$, the two procedures have the same running time and the same amount of memory used, to within 1%. This is more surprising, since procedure BDD-detection halts as soon as it finds a disjunct of $\Phi_C$ that is satisfied.

Predicate $\Phi'_S$ is also a disjunction, and here procedure BDD-detection does offer a significant benefit over procedure BDD-detection0. For example, for $N = 12$, the running times of BDD-detection and BDD-detection0 (with latency distribution $\rho_1$) are 3.03 sec and 1493.1 sec, respectively, so BDD-detection is 493 times faster than BDD-detection0. More generally, the ratio of the running times appears to grow exponentially with $N$. The ratio of the amount of memory used by the two procedures shows similar behavior. For example, for $N = 12$, the ratio of memory used is $(205MB/3.54MB) \approx 58$.

Thus, the optimization incorporated in procedure BDD-detection has drastically different effectiveness on different examples. Future work is needed to characterize the class of examples for which the optimization is effective.

# References

[AV97] Sridhar Alagar and S. Venkatesan. Techniques to tackle state explosion in global predicate detection. Submitted to *IEEE Transactions on Software Engineering*, 1997. Preliminary version appeared in *International Conference on Parallel and Distributed Systems* (ICPDS'94), pp. 412–417, 1994.

[BDD] The BDD Library (version 1.0). Available via http://www.cs.cmu.edu/ modelcheck/bdd.html.

[Bry92] R.E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3), 1992.

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

[FR94a] Eddy Fromentin and Michel Raynal. Inevitable global states: a concept to detect unstable properties of distributed computations in an observer independent way. In *Proc. Sixth IEEE Symposium on Parallel and Distributed Processing*, 1994.

[FR94b] Eddy Fromentin and Michel Raynal. Local states in distributed computations: A few relations and formulas. *Operating Systems Review*, 28(2), April 1994.

[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.

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

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

[Lyn96] Nancy A. Lynch. *Distributed Algorithms.* Morgan Kaufmann, 1996.

[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 International Workshop on Distributed Algorithms (WDAG '91)*, volume 579 of *Lecture Notes in Computer Science*, pages 254–272. Springer-Verlag, 1991.

[Sto97] Scott D. Stoller. Detecting global predicates in distributed systems with clocks. In Marios Mavronikolas, editor, *Proc. 11th International Workshop on Distributed Algorithms (WDAG '97)*, volume 1320 of *Lecture Notes in Computer Science*, pages 185–199. Springer-Verlag, 1997.

[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. Appeared as ACM SIGPLAN Notices 28(12), December 1993.

# Appendix: Pseudo-code for Example Protocols

## Pseudo-code for Coherence Protocol

The state variables (with initial values) of each process are:

| | |
|---|---|
| bool $\mathit{rdg} =$ false, $\mathit{wrtg} =$ false | whether process is reading or writing, respectively |
| bool $\mathit{wtgToRead} =$ false, | whether process is waiting to read |
| bool $\mathit{wtgToWrite} =$ false | whether process is waiting to write |
| PID set $\mathit{writeReq} = \emptyset$ | write requests that haven't been OK'd yet. |
| PID set $\mathit{OKsent} = \emptyset$ | processes we sent WriteReq to and didn't receive WriteDone from |
| PID set $\mathit{OKrcvd} = \emptyset$ | processes we received WriteOK from (valid when *wtgToWrite*=true). |
| float $\mathit{timer} =$ expRand(4) | each timer is initially set to a random value |

where $T$ set is the type of subsets of $T$.

Pseudo-code for the protocol appears in Figures 3 and 4. In pseudo-code, we represent the timer as a (special) variable *timer*; thus, a process sets its timer (or re-sets its timer, if its timer is already running) simply by assigning to the variable *timer*. For compactness, we use indentation to indicate the block structure of the program. In Figure 3, $<_{lex}$ denotes lexicographic order on vector timestamps, and $\mathrm{rand}_{\mathrm{bool}}()$ returns a random bit.

In the buggy version of the protocol, WriteOK is included with every WriteDone, *i.e.*, on line (*) in Figure 3, $\{\mathrm{WriteDone}\}$ is replaced with $\{\mathrm{WriteDone}, \mathrm{WriteOK}\}$.

```
Process $i$ : On timerExpire :
if rdg then
    // Stop reading.
    rdg := false
    send {WriteOK} to each member of writeReq
    OKsent := writeReq
    writeReq := $\emptyset$
    Set timer to start reading/writing again later.
    timer := expRand(4)
else if wrtg then
    // Stop writing. Send WriteDone to everyone and WriteOK to writeReq.
    wrtg := false
    send {WriteDone, WriteOK} to each member of writeReq
    send {WriteDone} to each member of (PID $\setminus$ writeReq)                                  (*)
    OKsent := writeReq
    writeReq := $\emptyset$
    Set timer to start reading/writing again later.
    timer := expRand(4)
else // Try to start reading or writing (choose non-deterministically).
    if $\mathrm{rand}_{\mathrm{bool}}()$ then
        wtgToWrite := true
        // Send WriteReq to everyone.
        send {WriteReq} to each member of PID
        OKrcvd := $\emptyset$
        // Set writeReqVTS to equal the timestamp that will be on the outgoing WriteReq messages.
        writeReqVTS := $vc_i$ with component $i$ incremented by 1
    else
        if OKsent $= \emptyset$ then
            // Start reading immediately. Set timer to stop reading.
            timer := expRand(4)
        else wtgread := true
```

Figure 3: Pseudo-code for coherence protocol (part 1).

## Pseudo-code for Spanning-Tree Algorithm

The state variables (with initial values, when they matter) of each process are:

| | |
|---|---|
| bool *hasParent* = false | whether process has a parent yet |
| nborid *parent*, | parent (valid when *hasParent* = true) |
| int*level* = 0 | level in the spanning tree (for processes except 0: valid when *hasParent* = true) |

Initially, the timer of process 0 is set to a value returned by expRand(4); the timers of the other processes are not set. Note that process 0 always has *hasParent* = false and *level* = 0, and this value of *level* is always meaningful. The pseudo-code appears in Figure 5, where nbors($i$) $\subseteq$ PID denotes the set of neighbors of process $i$.

In the buggy version of the algorithm, the conjunct $i \neq 0$ is omitted from the pseudo-code.

```
Process $i$ : On receiving $msg$ from $j$ :
if WriteReq $\in msg$ then
    if $rdg \lor wrtg \lor (wtgToWrite \land writeReqVTS <_{lex} \mathrm{ts}(m))$ then
        insert $j$ in $writeReq$
    else
        send {WriteOK} to $j$
        insert $j$ in $OKsent$
if WriteOK $\in msg$ then
    insert $j$ in $OKrcvd$
    if $OKrcvd = (\mathrm{PID} \setminus \{i\})$ then
        // Start writing. Set timer to stop writing later.
        $wtgToWrite$ := false
        $wrtg$ := true
        $timer$ := expRand(4)
if WriteDone $\in msg$ then
    remove $j$ from $OKsent$
    if $(wtgToRead \land OKsent = \emptyset)$ then
        // Start reading. Set timer to stop reading later.
        $wtgToRead$ := false
        $rdg$ := true
        $timer$ := expRand(4)
```

Figure 4: Pseudo-code for coherence protocol (part 2).

```
Process $i$ : On timerExpire :
// Only process 0 uses its timer.
send $level$ to each member of nbors($i$)

Process $i$ : On receiving $\ell$ from $j$ :
if $(\neg hasParent \land i \neq 0)$ then
    $hasParent$ := true
    $parent$ := nborOfPID($i, j$)
    $level$ := $\ell + 1$
    send $level$ to each member of nbors($i$) $\setminus \{parent\}$
```

Figure 5: Pseudo-code for spanning tree algorithm.