Efficient Symbolic Detection of Global Properties in Distributed Systems

Scott D. Stoller and Yanhong A. Liu

Computer Science Dept., Indiana University, Bloomington, IN 47405, USA
{stoller,liu}@cs.indiana.edu.

Abstract. A new approach is presented for detecting whether a computation of an asynchronous distributed system satisfies **Poss** Φ (read "possibly Φ"), meaning the system could have passed through a global state satisfying property Φ. 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. In an asynchronous distributed system,[1] 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 (*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, 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 c are exactly 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 **Poss** Φ iff, in *some* history consistent with c, the system passes through a global state satisfying Φ. A computation c satisfies **Def** Φ iff, in *all* histories consistent with c, the system passes through a global state satisfying Φ.

Cooper and Marzullo give centralized algorithms for detecting **Poss** Φ and **Def** Φ for an arbitrary predicate Φ [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. **Poss** Φ and **Def** Φ are evaluated by straightforward traversals of the lattice.

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 comes from the (exponential) number of CGSs of a computation in which there is little communication. Any detection algorithm that enumerates all CGSs—like the algorithms in [CM91,MN91,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 is to restrict the problem and develop efficient algorithms for detecting only certain classes of predicates [GW94,TG93].[2] Another approach is to modify

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

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

some aspect of the problem—for example, detecting a different modality [FR97] or assuming that the system is partially synchronous [MN91,Sto97].

This paper presents an efficient and general approach to detecting **Poss** Φ. In this approach, the set of CGSs is represented symbolically, using boolean formulas implemented as ordered binary decision diagrams (BDDs), and **Poss** Φ is detected by testing satisfiability of a formula. 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 detection algorithm. Section 4 gives performance results from using the new algorithm and (for comparison) 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. Both methods greatly reduce memory consumption, though at a significant cost in running time. Section 5 compares our work to temporal-logic model checking. Directions for future work include extending our algorithm to support on-line detection, applying our symbolic approach to detection of **Def** Φ, and experimenting with the use of a satisfiability checker, such as tableau [CA96], instead of BDDs. Our approach does not involve computation of fixed points, so the use of a canonical form, such as BDDs, is not essential.

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 timer expiration event of process p, component p of vc_p 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 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 the value of the process's vector clock.

[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.

2.2 Consistent Global States and Poss Φ

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). 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 s_1 and s_2 of a computation are *concurrent*, denoted $s_1 \parallel s_2$, iff neither happened-before the other: $s_1 \parallel s_2 \triangleq s_1 \not\rightarrow s_2 \ \wedge \ s_2 \not\rightarrow s_1$. 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,

$$\begin{aligned}&(\forall i_1 \in \mathrm{dom}(c[p_1]) : (\forall i_2 \in \mathrm{dom}(c[p_2]) : \\ &\quad c[p_1][i_1] \rightarrow c[p_2][i_2] \ \equiv \ c[p_1][i_1](vc_{p_1}) \prec c[p_2][i_2](vc_{p_2})))\end{aligned} \tag{1}$$

where for a sequence σ, $\mathrm{dom}(\sigma) = \{0, 1, \ldots, (|\sigma| - 1)\}$, where $|\sigma|$ is the length of σ. 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 p_1 and a local state s_2 of process p_2,

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

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

Now we define **Poss**. A computation c satisfies **Poss** Φ, denoted $c \models$ **Poss** Φ, iff there exists a consistent global state of c that satisfies Φ.

3 Detection Method

To test $c \models$ **Poss** Φ efficiently using symbolic methods, we generate a formula b such that b is satisfiable iff $c \models$ **Poss** Φ. 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 $\boldsymbol{x}$ denote the collection of variables $x_0, x_1, \ldots, x_{N-1}$, and let $\boldsymbol{vc}$ 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

$$\Phi(\boldsymbol{x}) \ \wedge \ \mathrm{globalState}_c(\boldsymbol{x}, \boldsymbol{vc}) \ \wedge \ \mathrm{consis}_c(\boldsymbol{vc}) \tag{3}$$

where

$$\mathrm{globalState}_c(\boldsymbol{x}, \boldsymbol{vc}) = \bigwedge_{p \in \mathrm{PID}} \ \bigvee_{i \in \mathrm{dom}(c[p])} x_p = c[p][i](x_p) \ \wedge \bigwedge_{q \in \mathrm{PID}} vc_{p,q} = c[p][i](vc_p)[q]$$

$$\mathrm{consis}_c(\boldsymbol{vc}) = \bigwedge_{p_1 \in \mathrm{PID}} \ \bigwedge_{p_2 \in (\mathrm{PID} \setminus \{p_1\})} vc_{p_2,p_1} \leq vc_{p_1,p_1}$$

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 $\mathbf{Poss}(y_0 + y_1 = 1)$. 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

$$(y_0 + y_1 = 1) \;\wedge\; \text{globalState}_c(y_0, y_1, \boldsymbol{vc}) \;\wedge\; \text{consis}_c(\boldsymbol{vc}) \tag{4}$$

where

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

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 change 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:

$$\Phi(\boldsymbol{x}) \;\wedge\; \text{globalState}_c(\boldsymbol{x}, \boldsymbol{idx}) \;\wedge\; \text{consis}_c(\boldsymbol{idx}) \tag{5}$$

where

$$\text{globalState}_c(\boldsymbol{x}, \boldsymbol{idx}) = \bigwedge_{p \in \text{PID}} \; \bigvee_{i \in \text{dom}(c[p])} x_p = c[p][i](x_p) \;\wedge\; idx_p = i \tag{6}$$

$$\text{consis}_c(\boldsymbol{idx}) = \bigwedge_{\substack{p_1 \in \text{PID} \\ p_2 \in (\text{PID} \setminus \{p_1\})}} \; \bigvee_{i_2 \in \text{dom}(c[p_2])} idx_{p_2} = i_2 \;\wedge\; c[p_2][i_2](vc_{p_2})[p_1] \le idx_{p_1} \tag{7}$$

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

Revisiting the above example, we obtain, instead of (4), the formula

$$(y_0 + y_1 = 1) \;\wedge\; \text{globalState}_c(y_0, y_1, \boldsymbol{idx}) \;\wedge\; \text{consis}_c(\boldsymbol{idx}) \tag{8}$$

where

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

3.1 Implementation and an Optimization

We represent the formula defined by (5) 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 many formulas of interest, BDDs are more compact than other canonical forms (such as conjunctive normal form). Let true_{bdd} and $\text{false}_{\text{bdd}}$ denote the BDDs representing true and false, respectively. Let $\wedge_{\text{bdd}}$ denote conjunction of BDDs. Let a formula with an overline denote a function that returns the BDD representation of that formula. Formula b is constructed and tested for satisfiability by procedure BDD-detection0 in Figure 1.

The functions $\overline{\text{globalState}}$ and $\overline{\text{consis}}$ are easily written based on (6) and (7). The numbers in vector timestamps are encoded as unsigned integers, with a binary variable representing each bit; the number of

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

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 Φ.

```
procedure BDD-detection0(c, Φ̄)
    b := true_bdd
    b := b ∧_bdd globalStatē_c(x, idx)
    b := b ∧_bdd consis̄_c(idx)
    b := b ∧_bdd Φ̄(x)
    if b = false_bdd then
        return("c ⊭ Poss(Φ)")
    else return("c ⊨ Poss(Φ)")
```

```
procedure BDD-detection(c, ⋁_{α∈S} Φ̄_α)
    b := true_bdd
    b := b ∧_bdd globalStatē_c(x, idx)
    b := b ∧_bdd consis̄_c(idx)
    for each α in S
        b1 := b ∧_bdd Φ̄_α(x)
        if b1 ≠ false_bdd then
            return("c ⊨ Poss(Φ)")
    return("c ⊭ Poss(Φ)")
```

Fig. 1. Pseudo-code for BDD-detection0 and BDD-detection.

Often (as in both examples in Section 4), Φ is a disjunction: $\Phi = \bigvee_{\alpha \in S} \Phi_\alpha$, for some set S. Procedure BDD-detection0 can be optimized by distributing the conjunctions over the disjunction, yielding procedure BDD-detection in Figure 1. By testing each disjunct of Φ separately, BDD-detection avoids constructing the potentially large intermediate result $\overline{\Phi}$.

4 Examples

We compare the performance of 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 a depth-first-search search of the lattice of CGSs. 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 consider cases where $c \models \mathbf{Poss}\,\Phi$ holds and cases where it doesn't. The most common use of detection algorithms for **Poss** is to check that an invariant I holds, by detecting whether the computation satisfies $\mathbf{Poss}\,\neg I$. So, we consider correct and buggy versions of each example protocol.

For each version of each example, we use a simulator to generate a computation, and then we analyze that computation using both BDD-detection and DFS-detection. By default, the simulator selects message delays from the distribution $\rho_1 = 1 + \text{expRand}(1)$, where $\text{expRand}(\mu)$ denotes an exponential distribution with mean μ. To measure the sensitivity of the analysis cost to message latencies, we consider 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 algorithms we measured are sequential, so the use of a parallel machine was irrelevant. We use the BDD library developed by E. M. Clarke's group at CMU [BDD]. The reported running times are "user times" obtained from the UNIX `time` command; thus, they reflect the CPU time consumed.

For BDD-detection, the variable ordering can affect performance. The overall variable ordering is $x_0, x_1, \ldots, x_{N-1}, idx_0, idx$ where x_p denotes the sequence of binary variables encoding the local state of process p excluding idx_p and excluding variables not mentioned in the predicate being detected, and idx_p denotes the sequence of binary variables encoding the "index" of the local state.

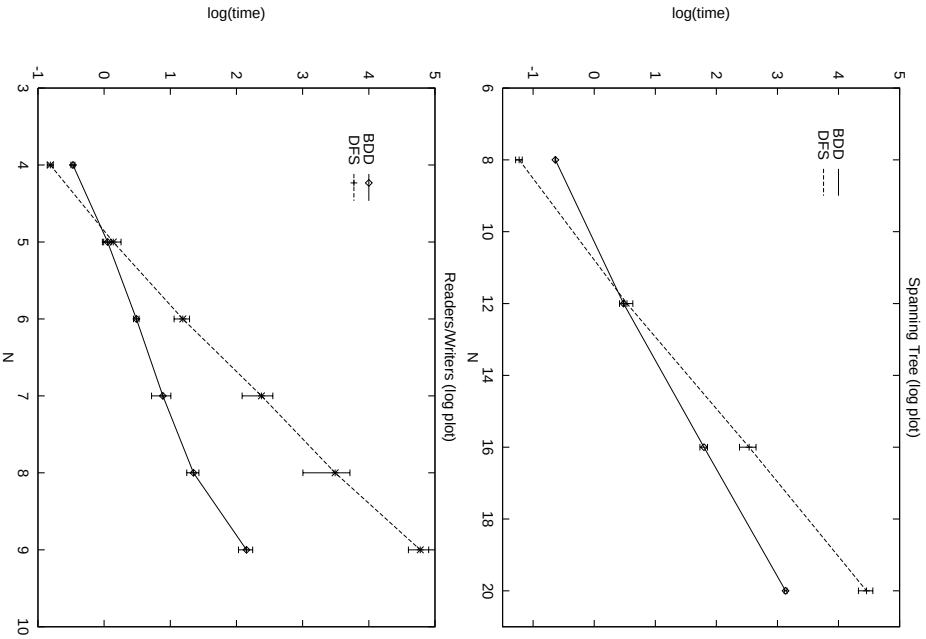

Fig. 2. Left: Logarithm of running time of detection algorithms on coherence protocol. Right: Logarithm of running time of detection algorithms on spanning-tree algorithm.

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 the following invariant: when one process is writing, no other process is reading or writing. Formally, we detect the predicate

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

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 left graph in Figure 2 shows $\log_{10}(t_{\mathrm{BDD}}(N))$ and $\log_{10}(t_{\mathrm{DFS}}(N))$ for the coherence protocol, where $t_{\mathrm{BDD}}(N)$ and $t_{\mathrm{DFS}}(N)$ denote the average running times, in seconds, of BDD-detection and DFS-detection, respectively. The average is over 10 different seeds of the random number generator; the error bars show the standard deviation. These functions both exhibit exponential growth—not surprising, since the number of CGSs is exponential in N. Nevertheless, for larger values of N, the difference in the running times of the two procedures is dramatic. For example $t_{\mathrm{DFS}}(9)/t_{\mathrm{BDD}}(9) \approx 433$; 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 behavior also occurs with latency distribution ρ_0.

Now consider the buggy coherence protocol. We ignore computations in which the bug does not manifest itself in a violation of Φ_C. BDD-detection is again faster than DFS-detection, though by a smaller margin—for example, by a factor of 46 at $N = 9$. The running time of BDD-detection is roughly independent of whether $c \models \mathbf{Poss}\,\Phi_C$ holds. In contrast, the average running time of DFS-detection is reduced by a factor of 7 to 10 when $c \models \mathbf{Poss}\,\Phi_C$ 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.

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, 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 RC corresponds to these relative coordinates. For a process p and relative coordinate r, $\mathrm{PIDofRC}(p, r)$ 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 $\mathrm{RCofPID}(p, q)$ is the relative coordinate of q with respect to p. Thus, $\mathrm{PIDofRC}(p, \mathrm{RCofPID}(p, q)) = q$.

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 that implies absence of cycles: the level of a process is larger than the level of its parent. Formally, we detect the predicate

$$\Phi_S = \bigvee_{p \in \mathrm{PID}} hasParent_p \ \wedge \ level_p \leq level_{\mathrm{PIDofRC}(p, parent_p)}$$

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

where boolean variable $hasParent_p$ indicates whether process p has gotten a parent, $parent_p$ is the (relative coordinate of) the parent of process p, and $level_p$ is the level of process p in the spanning tree.

Φ_S cannot be expressed directly as a boolean formula using the given variables, because $level_{p,\mathrm{PIDofRC}(parent_p)}$ is not a particular variable. So, we use DFS-detection to detect Φ_S but use BDD-detection to detect the following logically equivalent predicate:

$$\Phi'_S = \bigvee_{p \in \mathrm{PID}} \bigvee_{r \in \mathrm{RC}} hasParent_p \ \wedge \ parent_p = r \ \wedge \ level_p \leq level_{\mathrm{PIDofRC}(p,r)}.$$

We analyze computations of this algorithm in a network with a grid topology. Each row in the grid contains $m = \lfloor \sqrt{N} \rfloor$ processes. Each process is connected to its neighbors in the grid. Thus, process i is connected to processes $i-1$ (if $i > 0$), $i+1$ (if $i < N-1$), $i-m$ (if $i \geq m$), and $i+m$ (if $i < N-m$).

The right graph in Figure 2 shows $\log_{10}(t_{\mathrm{BDD}}(N))$ and $\log_{10}(t_{\mathrm{DFS}}(N))$ for the spanning-tree algorithm. Again, the average is over 10 different seeds of the random number generator, and the error bars show the standard deviation. BDD-detection is significantly faster for larger values of N; for example, $t_{\mathrm{DFS}}(20)/t_{\mathrm{BDD}}(20) = (28797.6 \text{ sec}/1360.2 \text{ sec}) \approx 21.2$. The ratio $t_{\mathrm{DFS}}(N)/t_{\mathrm{BDD}}(N)$ again increases exponentially with N. This behavior also occurs with latency distribution ρ_0.

For the buggy spanning-tree algorithm, DFS-detection is much faster than BDD-detection when $\mathbf{Poss}\,\Phi_S$ holds and is much slower than BDD-detection when $\mathbf{Poss}\,\Phi_S$ does not hold. The running time of the BDD algorithm is again roughly independent of whether $c \models \mathbf{Poss}\,\Phi_S$ holds (for $4 \leq N \leq 20$, the running time for seeds that cause $\mathbf{Poss}\,\Phi'_S$ to hold is within a factor of 2 of the running time for other seeds). In contrast, when $\mathbf{Poss}\,\Phi_S$ holds, DFS-detection is "lucky" and finds a CGS satisfying Φ_S very early in the search: for $4 \leq N \leq 20$, DFS-detection is approximately 10^5 times faster when $c \models \mathbf{Poss}\,\Phi_S$ than when $c \not\models \mathbf{Poss}\,\Phi_S$.

We also implemented the spanning-tree algorithm using PIDs rather than relative coordinates to indicate a process's parent. The effect on the running time of DFS-detection is negligible. The memory usage and running time of BDD-detection increase by roughly the same percentage as the number of bits per global state (which is the number of variables in the BDD), e.g., for $N = 20$, by approximately 20%.

4.5 Memory Usage

BDD-detection uses significantly more memory than DFS-detection, because DFS-detection never stores any representation of the entire set of CGSs. 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 2.6 MB at $N = 9$. For the spanning-tree example, the same asymptotic behavior 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 discussed next.

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]. Variables were reordered once, immediately after construction of $\overline{\mathrm{globalState}}_c(\boldsymbol{x}, \boldsymbol{idx}) \wedge_{\mathrm{bdd}} \overline{\mathrm{consis}}_c(\boldsymbol{idx})$. According to [BDD], the sift method "generally achieves greater size reductions, but is slower" than window3.

For the coherence protocol, the window method is preferable, because the increase in running time is smaller (typically a factor of about 1.5, compared to a factor of about 4 for sift) and, unexpectedly, the the decrease in memory usage is greater (typically a factor in the range 0.2–0.4, compared to 0.3–0.5 for sift). For the spanning-tree example, the sift method is preferable, because the decrease in memory usage is greater (e.g., a factor of 0.05 at $N = 9$, compared to 0.08 for window) and, unexpectedly, the increase in running time is smaller (typically a factor of about 6, compared to 9 for window). For the spanning-tree example, the fractional reduction in memory usage increases with N.

4.7 Comparing Performance of BDD-detection and BDD-detection0

Predicates Φ_C and Φ'_S are disjunctions, so it is interesting to compare procedures BDD-detection and BDD-detection0. For the correct and buggy coherence protocols, the two procedures have the same the running time and same amount of memory used, to within 1%. This is more surprising for the buggy protocol, because BDD-detection halts as soon as it finds a disjunct of Φ_C that is satisfied. For the spanning-tree algorithm, BDD-detection is significantly more efficient than BDD-detection0, with benefits that appear to grow exponentially with N. For example, for $N = 12$, so BDD-detection is 493 times faster than BDD-detection0 and uses 0.017 as much memory. Thus, the optimization incorporated in procedure BDD-detection has drastically different effectiveness on different examples. Further work is needed to characterize the class of examples for which the optimization in BDD-detection is effective.

5 Comparison with Symbolic Model Checking for CTL

Detection of **Poss** Φ can be reduced to CTL model checking [C+92]: a computation is encoded as a transition system whose runs are the histories consistent with the computation, and a CTL model checker is used to check whether that transition system satisfies the CTL formula $\exists \diamond \Phi$. With this encoding, an BDD-based model checker, such as SMV [SMV], would represent sets of CGSs as BDDs, as we do. However, that approach could still differ appreciably in performance from our algorithm, because different intermediate BDDs would be constructed. For example, with our method, the iterative calculations in the construction of $\overline{\text{globalState}}$ and $\overline{\text{consis}}$ are independent of Φ. With SMV, the corresponding iterative fixed-point calculation used to evaluate $\exists \diamond \Phi$ depends on Φ (roughly, the effect is as if lines 2 and 4 were swapped in BDD-detection), which might make the BDDs obtained in each iteration larger. Further experiments are needed to determine the performance impact of such differences.

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 (ver. 1.0). 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.

[C+92] E. M. Clarke et al. Automatic verification of sequential circuit design. In C. A. R. Hoare and M. J. C. Gordon, editors, *Mechanized Reasoning and Hardware Design*. Prentice-Hall, 1992.

[CA96] James M. Crawford and Larry D. Auton. Experimental results on the crossover point in random 3-SAT. *Artificial Intelligence*, 81(1):31–57, 1996.

[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.

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

[FR97] Eddy Fromentin and Michel Raynal. Inevitable global states: a concept to detect unstable properties of distributed computations in an observer independent way. *Journal of Computer and System Sciences*, 55(3), Dec. 1997.

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

[SMV] SMV. http://www.cs.cmu.edu/ modelcheck/smv.html.

[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. 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 rdg = false, $wrtg$ = false	whether process is reading or writing, respectively
bool $wtgToRead$ = false,	whether process is waiting to read
bool $wtgToWrite$ = false	whether process is waiting to write
PID set $writeReq = \emptyset$	write requests that haven't been OK'd yet.
PID set $OKsent = \emptyset$	processes we sent WriteReq to and didn't receive WriteDone from
PID set $OKrcvd = \emptyset$	processes we received WriteOK from (valid when $wtgToWrite$=true).
float $timer$ = 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 $\text{rand}_{\text{bool}}()$ returns a random bit.

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

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 timerExpire :
if rdg then
    // Stop reading.
    rdg := false
    send {WriteOK} to each member of writeReq
    OKsent := writeReq
    writeReq := ∅
    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 \ writeReq)                    (*)
    OKsent := writeReq
    writeReq := ∅
    Set timer to start reading/writing again later.
    timer := expRand(4)
else // Try to start reading or writing (choose non-deterministically).
    if rand_bool() then
        wtgToWrite := true
        // Send WriteReq to everyone.
        send {WriteReq} to each member of PID
        OKrcvd := ∅
        // 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 = ∅ then
            // Start reading immediately. Set timer to stop reading.
            timer := expRand(4)
        else wtgread := true
```

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

```
Process i : On receiving msg from j :
if WriteReq ∈ msg then
    if rdg ∨ wrtg ∨ (wtgToWrite ∧ writeReqVTS <_lex ts(m)) then
        insert j in writeReq
    else
      send {WriteOK} to j
      insert j in OKsent
if WriteOK ∈ msg then
    insert j in OKrcvd
    if OKrcvd = (PID \ {i}) then
      // Start writing. Set timer to stop writing later.
      wtgToWrite := false
      wrtg := true
      timer := expRand(4)
if WriteDone ∈ msg then
    remove j from OKsent
    if (wtgToRead ∧ OKsent = ∅) then
      // Start reading. Set timer to stop reading later.
      wtgToRead := false
      rdg := true
      timer := expRand(4)
```

Fig. 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 ℓ from j :
if (¬hasParent ∧ i ≠ 0) then
    hasParent := true
    parent := RCofPID(i, j)
    level := ℓ + 1
    send level to each member of nbors(i) \ {parent}
```

Fig. 5. Pseudo-code for spanning tree algorithm.