Exploiting Predicate Structure For Efficient Reachability Detection

Sujatha Kashyap
ECE Department
University of Texas at Austin
Austin, TX 78712, USA
kashyap@ece.utexas.edu

Vijay K. Garg
ECE Department
University of Texas at Austin
Austin, TX 78712, USA
garg@ece.utexas.edu

ABSTRACT

Partial order (p.o.) reduction techniques are a popular and effective approach for tackling state space explosion in the verification of concurrent systems. These techniques generate a reduced search space that could be exponentially smaller than the complete state space. Their major drawback is that the amount of reduction achieved is highly sensitive to the properties being verified. For the same program, different properties could result in *very* different amounts of reduction achieved.

We present a new approach which combines the benefits of p.o. reduction with the added advantage that the size of the constructed state space is completely independent of the properties being verified. As in p.o. reduction, we use the notion of *persistent sets* to construct a representative interleaving for each maximal trace of the program. However, we retain concurrency information by assigning vector timestamps to the events in each interleaving. Our approach hinges upon the use of efficient algorithms that parse the encoded concurrency information in the representative interleaving to determine whether a safety violation exists in *any* interleaving of the corresponding trace. We show that, for some types of predicates, reachability detection can be performed in time that is polynomial in the length of the interleaving. Typically, these predicates exhibit certain characteristics that can be exploited by the detection algorithm.

We implemented our algorithms in the popular model checker SPIN, and present experimental results that demonstrate the effectiveness of our techniques. For example, we verified a distributed dining philosophers protocol in 0.03 seconds, using 1.253 MB of memory. SPIN, using traditional p.o. reduction techniques, took 759.71 seconds and 439.116 MB of memory.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Program Verification

ASE'05, November 7–11, 2005, Long Beach, California, USA.

General Terms

Verification

Keywords

model checking, reachability, Mazurkiewicz traces

1. INTRODUCTION

State space enumeration using interleaving semantics is a highly successful strategy used in the verification of software systems. In this approach, the software system is represented by a labeled transition graph in which every reachable state of the system is enumerated. However, the size of the constructed graph, also known as the state space, quickly becomes exponential in the size of the program description. This is known as *state space explosion*, and is the fundamental obstacle in the verification of large software systems. Concurrent systems are particularly vulnerable to state space explosion because of the combinatorial number of possible interleavings of a set of concurrent events. For instance, a set of n concurrent events can be interleaved in $n!$ ways, and an exhaustive state space search would explore each of these interleavings.

Partial order (p.o.) reduction strategies [24, 18, 19, 11] have emerged as a highly successful, and widely deployed, technique to reduce state space explosion. The widely-used model checker, SPIN, employs p.o. reduction.

Partial order reduction is based on the observation that it is not always necessary to explore *all* possible interleavings of a set of concurrent events. Sometimes, a given property is satisfied in one interleaving if and only if it is satisfied in all interleavings. Such properties are called *equivalence-robust* [19]. For the verification of equivalence-robust properties, it is sufficient to explore a single representative interleaving of a set of concurrent events. However, safety properties are rarely equivalence-robust. For example, in a mutual exclusion algorithm, a safety property of interest is that no two processes should be in the critical section simultaneously, that is, $\neg EF(incs_i \wedge incs_j)$. Consider the (improbably) naive implementation of mutual exclusion in Figure 1, where each process blindly enters and exits the critical section. The interleaving in Figure 1(b) satisfies the safety property, whereas a safety violation occurs in Figure 1(c).

To deal with more general properties, p.o. reduction techniques exploit the notion of *invisibility* of events. An event is said to be invisible with respect to a predicate only if execution of the event has no effect on the value of the pred-

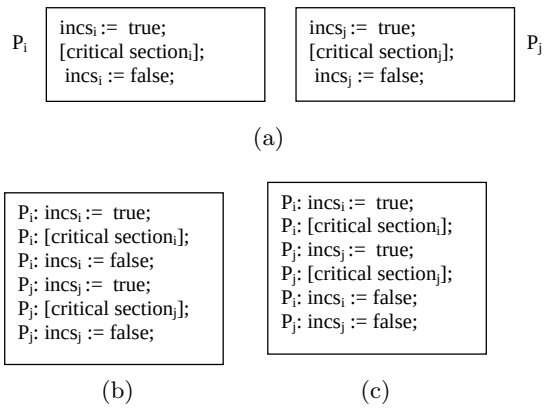

Figure 1: A naive distributed mutual exclusion implementation. (a) Concurrent processes P_i and P_j, (b) a safe interleaving of events, and (c) an unsafe interleaving.

icate. Let each state in an interleaving sequence be labeled by the value of the predicate to be checked. A sequence of adjacent states with the same label is called a *stutter*. A labeled interleaving sequence can be *collapsed* by replacing each stutter by a single state with the same label as the states in the stutter. For example, the labeled sequence *abbcccd* can be collapsed into the sequence *abcd*. Two interleaving sequences are said to be *stuttering-equivalent* iff they can be collapsed into the same labeled sequence. It has been shown [20] that properties expressible in temporal logics without the next-time operator (LTL_{-X}, CTL_{-X}) cannot distinguish between stuttering-equivalent paths. The reduced state space graph generated by p.o. reduction includes at least one stuttering-equivalent sequence for each interleaving sequence in the full state space graph.

Two distinct sequences can be stuttering-equivalent only if they contain invisible events. So, the effectiveness of p.o. reduction is highly dependent on the number of invisible events in the program. This makes p.o. reduction very sensitive to the predicates being checked [10]. If a large proportion of events change the value of the predicate, then little to no reduction may be achieved in the size of the constructed state space. Experimental results [10] show that the effectiveness of p.o. techniques diminishes *rapidly* with an increase in the number of visible events.

In this paper, we propose a new approach for detecting safety properties in a concurrent software system. Our approach achieves at least as much state space reduction as p.o. techniques. In addition, the amount of reduction achieved is independent of the properties being checked. Essentially, our approach eliminates the need to consider the visibility of events while constructing the reduced state space.

With p.o. reduction techniques, once a representative interleaving is chosen for inclusion in the reduced graph, we lose all information about the concurrency between events in the interleaving. In our proposed approach, we use p.o. techniques to generate a single representative interleaving for a set of concurrent events, while retaining concurrency information by assigning vector timestamps to the events in an interleaving. The vector timestamps allow us to treat the representative interleaving as a partially ordered set (poset) of events, in which concurrent events are incomparable. Thus, vector timestamps provide a "bridge" between interleaving semantics and a partial order *trace* semantics as described by Mazurkiewicz[16].

We also present algorithms that can use the concurrency information carried by the representative interleaving to determine whether *any* possible interleaving of the corresponding trace could result in a safety violation. We focus on detecting temporal properties of the form $EF(\psi)$. In general, the problem of detecting $EF(\psi)$ on a trace is NP-complete in the number of events in the trace (*i.e.*, in the length of the representative interleaving) [2]. However, for predicate classes that exhibit certain structure (characteristics), we can detect safety violations in time that is polynomial in the number of events in the trace. We introduce some of these predicate classes, together with polynomial-time detection algorithms for them.

Model-checking is rife with intractable problems. It well-known that model-checking is PSPACE complete [3]. This complexity arises from a combination of factors, including concurrency between processes, local non-deterministic choice within a process, and the range of the variables used in the program. Different approaches attempt to minimize the impact of different factors. Our approach minimizes the impact of concurrency on the size of the constructed state space, and usually results in an exponentially smaller state space representation. In this regard, we are more effective than p.o. reduction techniques.

The most significant contribution of this paper is the notion of performing model checking by exploiting the *structure* (characteristics) of the properties to be checked. For predicates exhibiting certain characteristics, we can perform reachability checking in time and space that is polynomial in the size of the constructed state space representation. In reality, many properties of interest do exhibit characteristics that make them amenable to polynomial time detection.

To demonstrate the effectiveness of our techniques, we implemented our algorithms in the SPIN model checker [12]. We chose SPIN for our implementation because it is currently the most widely-used software verification tool, and is especially popular for the verification of concurrent software systems. We present experimental results from the verification of three well-known distributed algorithms: a dining philosophers algorithm, a mutual exclusion algorithm, and a leader election algorithm. In experiments, an implementation of our algorithms detected safety violations in a leader election protocol in 53.53 seconds, whereas SPIN with p.o. reduction took 547.41 seconds to detect the same violations.

The rest of this paper is organized as follows. In Section 2, we present necessary background information and definitions. Section 3 formalizes the definition of a trace cover, which is a (possibly infinite) set of traces that contain all the reachable system states. Section 4 presents an algorithm for constructing a (possibly infinite) representative interleaving for each trace in a trace cover. Section 5 describes the mechanism for assigning vector timestamps to the events in an interleaving, in order to obtain the partial order representation of the corresponding trace. In Section 6, we present an algorithm that generates a *finite* trace cover for finite-state systems. We introduce some tractable classes of predicates in Section 7, and present polynomial-time algorithms to detect such predicates. Experimental results are provided in Section 8. We discuss related work in Section 9, and finish with some concluding remarks in Section 10.

2. PRELIMINARIES

A program P is a triple (S, T, s_0) where S is a finite set of states, T is a finite set of operations, and $s_0 \in S$ is the initial state. Each state $s \in S$ is associated with a set of operations, called *enabled*(s), which contains *all* the operations that can be executed from s. An operation $t \in$ *enabled*(s) transforms the state s into a *unique* state $s' \in S$.

Each occurrence (execution) of an operation is called an *event*. In this paper, we sometimes use the term "event" to mean "the operation of which this event is an occurrence". The context will make clear which interpretation is the relevant one. We denote the unique state s' reached upon executing the event α from the state s by $s' := \alpha(s)$.

DEFINITION 1. *An* **interleaving sequence** *of a program P is a (finite or infinite) sequence of events $w = \alpha_1\alpha_2....$, such that:*

1. $\alpha_1 \in enabled(s_0)$, *and*
2. $\forall i \geq 1 : \alpha_{i+1} \in enabled(s_i)$, *where* $s_i := \alpha_i(s_{i-1})$.

The set of states $\{s_0, s_1, s_2....\}$ generated by the interleaving sequence w is denoted by $states(w)$. For a finite interleaving sequence w, the final state reached by the sequence is denoted by $fin(s_0, w)$.

DEFINITION 2. *[16, 19] An* **independence relation** $I \subseteq T \times T$ *is an irreflexive, symmetric relation such that $(\alpha, \beta) \in I$ iff $\forall s \in S$:*

- **Enabledness:** $\alpha \in enabled(s) \Rightarrow (\beta \in enabled(s) \Leftrightarrow \beta \in enabled(\alpha(s)))$, *and*
- **Commutativity:** $(\alpha, \beta \in enabled(s)) \Rightarrow (\alpha(\beta(s)) = \beta(\alpha(s)))$.

The enabledness condition states that execution of α does not affect the enabledness of β, and the commutativity condition states that executing α and β in either order results in the same state.

The **dependency relation** D is the reflexive, symmetric relation given by $D = (T \times T) \setminus I$. Two events are independent if their corresponding operations are independent.

Mazurkiewicz [16] defined an equivalence relation $\equiv_D$ between finite sequences of operations, where $\equiv_D$ is the smallest transitive relation that satisfies the following conditions, for all $u, v, w \in T^*$:

1. $v \equiv_D v$.
2. If $v = u_1\alpha\beta u_2$ and $w = u_1\beta\alpha u_2$ for some $u_1, u_2 \in T^*$ and $\alpha, \beta \in T$, such that $(\alpha, \beta) \in I$, then $v \equiv_D w$.

Informally, $v \equiv_D w$ iff v can be transformed into w by repeatedly commuting adjacent independent operations.

This definition was extended to infinite sequences in [14]. Let $Pref(v)$ denote the set of all finite prefixes of a (finite or infinite) sequence $v \in T^* \cup T^\omega$. We say that $v \preceq_D w$ for some $w \in T^* \cup T^\omega$ iff $\forall u_1 \in Pref(v) : \exists u_2 \in Pref(w) :: u_1 \equiv_D u_2$. Now, for infinite sequences $v, w \in T^\omega$, $v \equiv_D w$ iff $(v \preceq_D w) \wedge (w \preceq_D v)$.

DEFINITION 3. *A* **trace** *is an equivalence class of the relation $\equiv_D$ over the interleaving sequences of a program P.*

For the rest of this paper, we fix the dependency relation D, and refer to it implicitly. A trace σ is also denoted by $[s_0, v]$, where s_0 is the (common) initial state of each interleaving sequence in the trace, and v is any member sequence of σ. It is easily shown [11] that if $v \equiv_D w$, then $fin(s_0, v) = fin(s_0, w)$. We denote the unique final state of a trace σ by fin_σ. For a trace σ, $Pref(\sigma) \stackrel{def}{\equiv} \bigcup_{v \in \sigma} Pref(v)$ denotes the set of all prefixes of the trace.

The *concatenation* of two traces $\sigma = [\zeta, u]$ and $\sigma' = [\theta, v]$ is defined when $\theta = fin_\sigma$, as $\sigma.\sigma' \stackrel{def}{\equiv} [\zeta, uv]$. We say a finite trace σ is **subsumed** by a finite or infinite trace ρ, denoted by $\sigma \sqsubseteq \rho$ iff there exists a trace σ' such that $\rho = \sigma.\sigma'$. If σ' is not empty, then we say $\sigma \sqsubset \rho$. Traces σ and σ' are said to be **consistent** iff $\exists \ddot{\sigma} : (\sigma \sqsubseteq \ddot{\sigma}) \wedge (\sigma' \sqsubseteq \ddot{\sigma})$.

Note that any interleaving sequence of a trace contains the same set of events. We will use the notation σ_E to denote a trace σ with E as its set of events.

3. TRACE COVERS

Let v be an interleaving sequence. Every state $s \in states(v)$ is the final state of some finite prefix of v, that is, $\forall s \in states(v) : (\exists u \in Pref(v) :: fin(s_0, u) = s)$. Let $States(\sigma)$ be the set of all states generated by any interleaving sequence of the trace σ. We have:

$$States(\sigma) = \bigcup_{v \in \sigma} states(v) = \bigcup_{u \in Pref(\sigma)} fin(s_0, u) \qquad (1)$$

LEMMA 1. *Given traces σ, σ': $\sigma \sqsubseteq \sigma' \Rightarrow States(\sigma) \subseteq States(\sigma')$.*

PROOF. From the definition of $\sqsubseteq$, $v \in Pref(\sigma) \Rightarrow v \in Pref(\sigma')$. From (1), this implies that $s \in States(\sigma) \Rightarrow s \in States(\sigma')$. □

DEFINITION 4. *A set of traces Δ of a program $P = (S, T, s_0)$ is called a* **trace cover** *iff for every reachable state $s \in S$, there exists a trace $\sigma \in \Delta$ such that $s \in States(\sigma)$.*

Lemma 1 implies that it is sufficient to consider only traces that are maximal under the $\sqsubseteq$ relation when constructing a trace cover.

Mazurkiewicz [16] showed a correspondence between a trace and a partial order on the events in the trace, such that every linearization of the partial order is an interleaving sequence of the trace and vice-versa. Denote the partially-ordered set (poset) corresponding to a trace σ_E by $(E, \rightarrow)$. The relation $\rightarrow$ corresponds to Lamport's *happened-before* (causality) relation [15], and is given by:

DEFINITION 5. *The happened-before relation $\rightarrow$ on a trace $\sigma_E = [s_0, w]$ is the smallest transitive relation that satisfies:*

$$(\alpha, \beta) \in D \wedge (w = u\alpha v\beta w') \Rightarrow \alpha \rightarrow \beta$$

where $\alpha, \beta \in E$, and $u, v \in T^*$, $w' \in T^* \cup T^\omega$.

Thus, given a dependency relation and a representative sequence of a trace, we can construct the poset corresponding to that trace.

Given a poset $(E, \rightarrow)$, events $\alpha, \beta \in E$ are said to be incomparable (denoted $\alpha \parallel \beta$) iff $\alpha \not\rightarrow \beta$ and $\beta \not\rightarrow \alpha$. An (order) **ideal** of a poset $(E, \rightarrow)$ is a subset $G \subseteq E$ such that for any $e, f \in E$, $(f \in G) \wedge (e \rightarrow f) \Rightarrow (e \in G)$. We will use

the term "ideal of a trace σ_E" to mean "ideal of the poset $(E, \rightarrow)$".

Let $G \subseteq E$ be an ideal of the trace σ_E. G can be viewed as a trace $\sigma_G = (G, \rightarrow)$. If G is finite, then $\sigma_G \sqsubseteq \sigma_E$. It has been shown (*e.g.*, see [22, 8]) that every state in $States(\sigma_E)$ corresponds to the final state of some ideal $G \subseteq E$, and vice-versa.

4. REPRESENTATIVE INTERLEAVINGS

In the previous section, we noted that given a representative interleaving sequence of a trace σ_E and the dependency relation D, we can obtain the poset $(E, \rightarrow)$. Also, we are only interested in traces that are maximal under $\sqsubseteq$, because these contain all the states of the program. Thus, in order to construct a trace cover, we simply need the dependency relation D, and a representative interleaving sequence from each maximal trace of the program. It was shown in [14] that the set of sequences that satisfy the following constraint is exactly the set of interleaving sequences of traces that are maximal under the relation $\sqsubseteq$.

CONSTRAINT 1. *[19, 14] If an operation α is enabled at some state of an interleaving sequence π, then an operation that is dependent on α (possibly α itself) must occur later (or immediately) in π.*

If two traces are inconsistent, then they are subsumed by distinct maximal traces, that is, they cannot be represented by the same interleaving sequence. The following lemma helps us determine, in an online fashion, when two traces are inconsistent.

LEMMA 2. *Given a trace $\sigma = [s_0, v]$ and events α_1 and α_2 such that $(\alpha_1, \alpha_2) \in D$. If $\alpha_1, \alpha_2 \in enabled(fin_\sigma)$, then $\sigma.[fin_\sigma, \alpha_1]$ is not consistent with $\sigma.[fin_\sigma, \alpha_2]$.*

PROOF. Assume, for contradiction, that $\sigma.[fin_\sigma, \alpha_1]$ is consistent with $\sigma.[fin_\sigma, \alpha_2]$. By the definition of trace concatention, $\sigma.[fin_\sigma, \alpha_1] = [s_0, v\alpha_1]$ and $\sigma.[fin_\sigma, \alpha_2] = [s_0, v\alpha_2]$. By the definition of consistent traces, there must exist a trace ρ such that $[s_0, v\alpha_1] \sqsubseteq \rho$ and $[s_0, v\alpha_2] \sqsubseteq \rho$. By the definition of the relation $\sqsubseteq$, there exists an interleaving sequence $w_1 \in \rho$ such that $v\alpha_1$ is a prefix of w_1. Similarly, there exists an interleaving sequence $w_2 \in \rho$ such that $v\alpha_2$ is a prefix of w_2. Since $[s_0, w_1] \equiv [s_0, w_2]$, each sequence must contain the same events. Thus, α_1 also occurs in w_2 and α_2 also occurs in w_1. More precisely, α_1 occurs after α_2 in w_2, and α_2 occurs after α_1 in w_1. Now, $w_1 \equiv_D w_2$ iff w_2 can be obtained from w_1 by repeatedly commuting adjacent *independent* operations. Clearly, in order to transform w_1 into w_2, α_1 and α_2 must be commuted at some iteration, which implies that α_1 and α_2 are independent. This contradicts the assumption that $(\alpha_1, \alpha_2) \in D$. □

DEFINITION 6. *[11, 18] A set T of transitions enabled in a state s is* **persistent** *in s iff, for any non-empty path starting from s in the full state space graph:*

$$s = s_1 \xrightarrow{t_1} s_2 \xrightarrow{t_2} s_3 ... \xrightarrow{t_{n-1}} s_n \xrightarrow{t_n} s_{n+1}$$

such that $\forall i \in \{1...n\} : t_i \notin T$, t_i is independent of all transitions in T.

It was shown [11, 18] that in order to construct at least one interleaving sequence per maximal trace, it is sufficient to explore only a persistent set of transitions from a given state. To create the fewest number of representative interleavings, we aim to use the smallest persistent set possible. The following lemma helps to reduce the size of the persistent sets chosen.

LEMMA 3. *If T is persistent in s and $\exists \alpha \in T$ such that $\forall \beta \in T \setminus \{\alpha\}$: $(\alpha, \beta) \in I$, then $T \setminus \{\alpha\}$ is also persistent in s.*

PROOF. Immediate from the definition of persistent sets. □

As we are interested in using the smallest persistent sets, we only choose persistent sets that satisfy the following constraint.

CONSTRAINT 2. *If T is persistent in s, then $\forall \alpha \in T, \exists \beta \in T \setminus \{\alpha\} : (\alpha, \beta) \in D$.*

By Lemma 3, any persistent set that does not satisfy the above constraint can be transformed into one that does, simply by removing the transitions that are independent of all other transitions in the set. By Constraint 2, if the persistent set contains more than one event, then the events are interdependent. Thus, each event in a persistent set must belong to a distinct maximal trace.

Algorithm 1 constructs a representative interleaving sequence for each maximal trace. The algorithm performs a breadth-first state space traversal, in which only a persistent subset of the set of enabled events is explored at each state. In steps 10-13, the first event from the persistent set is added to the current trace. By Constraint 2 and Lemma 2, the remaining events in the persistent set necessarily belong to inconsistent traces, so in steps 14-20, new traces are created for these events. Note that every sequence that is dequeued in line 25 represents some *maximal* trace of the program. Clearly, Algorithm 1 terminates iff the input program P has no infinite execution. The algorithm is presented here merely for clarity of presentation, and as a first step towards the construction of a finite trace cover.

THEOREM 1. *1. Every sequence generated by Algorithm 1 is an interleaving sequence of the input program P.*

2. Assuming that every queued sequence is eventually explored, Algorithm 1 produces a trace-equivalent sequence for every interleaving sequence of the program P that satisfies Constraint 1.

PROOF. 1. Obvious from the BFS construction.

2. The proof is by construction [19]. Let $w = \alpha_0, \alpha_1 ...$ be an arbitrary interleaving sequence of P starting from some state s in the full state graph, such that w satisfies Constraint 1.

   (a) If $\alpha_0 \in persistent(s)$, then the algorithm constructs the prefix α_0 of w, and the construction proceeds inductively from the state $\alpha_0(s)$.

   (b) If $\alpha_0 \notin persistent(s)$, let w' be the maximal prefix of w such that w' contains no operations from $persistent(s)$. From the definition of persistent sets (Definition 6), all the events of w' are independent of all the events in $persistent(s)$. Since w satisfies Constraint 1, it must contain some event that is dependent on an event in $persistent(s)$.

Algorithm 1: trace_cover

```
input : A program P, with initial state s_0.
output: A set of interleaving sequences {τ_0, τ_1....}.
1  begin
2    τ_0 := ε /* the empty string */
3    enqueue(τ_0)
4    while queue is not empty do
5      τ := head_of_queue()
6      s := fin(s_0, τ)
7      while enabled(s) ≠ ∅ do
8        work_set := persistent(s)
9        τ_old := τ
10       if work_set ≠ ∅ then
11         let α ∈ work_set
12         τ := τ_old.α /* add one transition to
           the current sequence */
13         s := α(s)
14         work_set := work_set \ {α}
15         while work_set ≠ ∅ do
16           /* create new sequences for the
             remaining transitions */
17           let β ∈ work_set
18           τ_new := τ_old.β
19           work_set := work_set \ {β}
20           enqueue(τ_new)
21         endw
22       endif
23     endw
24     /* Sequence completed, remove from the
       queue */
25     dequeue()
26   endw
27 end
```

So, w' is a *proper* prefix of w. Let $w = w'\beta v$ (where v could be the empty string). Thus, β is in $persistent(s)$. Since β is independent of all the events in w', $w'\beta v$ can be transformed into $\beta w'v$ by commuting adjacent independent operations $|w'|$ times. That is, $\beta w'v$ is trace-equivalent to w and the algorithm constructs the prefix β of a path that is trace-equivalent to w. The construction proceeds inductively from the state $\beta(s)$.

□

Algorithm 1 constructs a representative sequence for each maximal trace σ_E of the program. The following section presents a method to convert this representative sequence into the poset $(E, \rightarrow)$.

5. POSET REPRESENTATION OF A TRACE

Let w be an interleaving sequence created by Algorithm 1. In this section, we present a mechanism that assigns a *timestamp* to each event in w. This timestamping mechanism is a generalization of Fidge [7] and Mattern's [8] "vector clocks". A timestamp is an integer vector of dimension n, where n is the number of processes in the program[1]. We denote the timestamp of an event α by $\alpha.\nu$. The i^{th} component of the vector $\alpha.\nu$ is denoted by $\alpha.\nu[i]$.

Given two n-dimensional vector timestamps, $\alpha.\nu$ and $\beta.\nu$, we compare them as follows.

$$\alpha.\nu < \beta.\nu \stackrel{def}{\equiv} (\forall i : \alpha.\nu[i] \leq \beta.\nu[i]) \wedge (\exists j : \alpha.\nu[j] < \beta.\nu[j])$$

Our timestamping algorithm imposes the following constraint on the program P. Let P consist of n processes, $P_1 ... P_n$.

CONSTRAINT 3. *Each event in the program P "belongs to" exactly one process. Let P_i be the process to which α belongs. We denote this by $\alpha \in P_i$. Furthermore, $\alpha, \beta \in P_i \Rightarrow (\alpha, \beta) \in D$.*

Constraint 3 is quite a natural assumption for distributed and parallel programs. We now describe an online mechanism for assigning timestamps to the events in a sequence. We assume that the empty sequence ε contains the empty event ϵ, and $\epsilon.\nu = [0, 0, 0...., 0]$. We assume that every event is dependent on ϵ. When an event α is concatenated to the sequence τ, it is assigned a timestamp as follows.

1. Let $\alpha \in P_i$, where $1 \leq i \leq n$.
2. Calculate the set $dep(\alpha)$, where:
$$dep(\alpha) = \{\beta | (\beta \in \tau) \wedge (\alpha, \beta) \in D\}$$
3. For all $j \in \{1...n\}$, set:
$$\alpha.\nu[j] := max\{\beta.\nu[j] | \beta \in dep(\alpha)\}$$
4. Set $\alpha.\nu[i] := \alpha.\nu[i] + 1$.

Let τ be a representative sequence of the trace σ_E. The following theorem shows how our timestamping mechanism captures the poset $(E, \rightarrow)$.

THEOREM 2. *Given a trace σ_E, and $\alpha, \beta \in E$:*

$$\alpha \rightarrow \beta \Leftrightarrow \alpha.\nu < \beta.\nu$$

PROOF. Straightforward from the definition of $\rightarrow$ and the timestamping procedure. □

6. FINITE TRACE COVERS

As noted earlier, Algorithm 1 can produce an infinite number of traces, each of infinite length. For finite-state programs, it is possible to obtain a finite number of traces, each of finite length, which encode all the reachable states of the program. In this section, we present a modified version of Algorithm 1 which produces such a *finite* trace cover.

Let $(E, \rightarrow)$ be the poset corresponding to the trace σ_E. The **principal ideal** of an event $e \in E$ is denoted by $\downarrow e$, and is given by:

$$\downarrow e \stackrel{def}{\equiv} \{f \in E | (f \rightarrow e) \vee (f = e)\}$$

Intuitively, $\downarrow e$ is the minimal set of events that *must* occur in any ideal that includes e.

LEMMA 4. *Let $G \subseteq E$ be any ideal of $(E, \rightarrow)$ such that $e \in G$. Then, fin_{σ_G} is reachable from $fin_{\sigma_{\downarrow e}}$.*

[1]If the program can dynamically create and destroy processes, then we can take n to be the maximum number of processes created.

PROOF. It is clear that $\sigma_{\downarrow e} \sqsubseteq \sigma_G$. From the definition of $\sqsubseteq$, we have $\sigma_{\downarrow e}.\sigma_{G\setminus\downarrow e} = \sigma_G$. That is, fin_{σ_G} is reachable from $fin_{\sigma_{\downarrow e}}$ via an (actually, every) interleaving sequence of $\sigma_{G\setminus\downarrow e}$. □

Lemma 4 was used by McMillan to construct a finite complete prefix for Petri Net unfoldings [17]. We use the same approach here to obtain a finite trace cover.

Let e and f be two events such that their principal ideals have the same final state, that is, $fin_{\sigma_{\downarrow e}} = fin_{\sigma_{\downarrow f}}$. By Lemma 4, any ideal containing an event e' such that $e \rightarrow e'$ corresponds to a state which is reachable from $\sigma_{\downarrow e}$, hence is also reachable from $\sigma_{\downarrow f}$. So, we need not add events e', such that $e \rightarrow e'$, to $\sigma_{\downarrow e}$, *as long as we add the corresponding events to* $\sigma_{\downarrow f}$. That is, e can be marked as a "cutoff event" [17]. McMillan showed that a sufficient condition for marking an event e as a cutoff event is the existence of an event f in *any* trace of the program, such that $fin_{\sigma_{\downarrow f}} = fin_{\sigma_{\downarrow e}}$, and $|\downarrow f| < |\downarrow e|$.

Algorithm 2 uses this notion of cut-off events to prune the sequences constructed by Algorithm 1. When an event $\alpha \in P_i$ is identified as a cut-off event in the sequence τ, further events from P_i are ignored when constructing larger sequences that contain τ as a prefix. The key difference between Algorithms 2 and 1 is that Algorithm 2 chooses its persistent sets from the set of *eligible* events at each state, whereas Algorithm 1 considers the set of *enabled* events. If an event α is identified as a cutoff event, then the corresponding process, denoted by $proc(\alpha)$, is marked as a cutoff process. This is done in lines 21-23 and 26-28. In line 7, only events belonging to non-cutoff processes are marked as *eligible* events. In line 9, the routine $persistent'(s)$ returns a persistent subset of the *eligible* events at state s. This persistent set is then explored further in lines 9-29.

Two sets of traces are said to be **state-equivalent** iff they have the same set of reachable states. For a given program, let Δ be the set of traces produced by Algorithm 1, and Υ be the set of traces produced by Algorithm 2.

THEOREM 3. *[17]* Δ *is state-equivalent to* Υ.

PROOF. It is evident that every state in Υ is a state in Δ. The proof of the converse is as follows. Let G be an ideal of some trace in Δ, but not an ideal of any trace in Υ. Then, G contains a cutoff event, e. Thus, there is another event f in some trace of Δ such that $|\downarrow f| < |\downarrow e|$, and $fin_{\sigma_{\downarrow e}} = fin_{\sigma_{\downarrow f}}$. Thus, Δ has an ideal $G' = \downarrow f \cup (G \setminus \downarrow e)$ such that $fin_{\sigma_{G'}} = fin_{\sigma_G}$. Note that $|G'| < |G|$ because $|\downarrow f| < |\downarrow e|$. If G' is also not an ideal of Υ then, by similar reasoning, there is another ideal G'' in Δ such that $fin_{\sigma_{G''}} = fin_{\sigma_{G'}} = fin_{\sigma_G}$, such that $|G''| < |G'| < |G|$. If G'' is also not in Υ, we iterate this procedure again. We cannot iterate infinitely because the order $<$ on the size of ideals is well-founded. Therefore, there must exist some ideal H in Υ that produces the same state as G. □

Thus, Algorithm 2 also produces a trace cover for the program P. It now remains to be shown that Υ is finite.

THEOREM 4. *(a) Every trace in* Υ *is of finite length. (b) There are a finite number of traces in* Υ.

PROOF. (a) Algorithm 2 produces interleaving sequences, from which we extract the corresponding poset. Let N be the total number of distinct states in the given finite state program. Let $w = \alpha_1\alpha_2....$ be an interleaving sequence, starting from the initial state s_0, produced by Algorithm 2. Consider the first $N+1$ events in this sequence. Since there are only N states, there exist events α_i and α_j in w, where $1 \leq i < j \leq N+1$, such that the state corresponding to the ideal $\downarrow \alpha_i$ is the same as the state corresponding to $\downarrow \alpha_j$. Also, since w is a linearization of the partial order on $\alpha_1, \alpha_2...$, and $i < j$, we have $|\downarrow \alpha_i| < |\downarrow \alpha_j|$. Thus, α_i would be recognized as a cutoff event. Thus, the length of any interleaving sequence cannot be more than $N+1$.

(b) Follows from (a) and the fact that $|enabled(s)|$ is finite for each state s. □

Algorithm 2: finite_trace_cover

```
   input : A program P, with initial state s_0.
   output: A finite set of finite interleaving sequences
           {τ_0, τ_1....}.
1  begin
2    τ_0 := ε /* the empty string */
3    enqueue(τ_0, ∅) /* no cutoff processes */
4    while queue is not empty do
5      (τ, cutoff_procs) := head_of_queue()
6      s := fin(s_0, τ)
7      eligible(s) := {γ|(γ ∈ enabled(s)) ∧ (proc(γ) ∉
       cutoff_procs)}
8      while eligible(s) ≠ ∅ do
9        work_set := persistent'(s)
10       τ_old := τ
11       if work_set ≠ ∅ then
12         let α ∈ work_set
13         τ := τ_old.α /* add one transition to
           the current sequence */
14         s := α(s)
15         work_set := work_set \ {α}
16         while work_set ≠ ∅ do
17           /* create new sequences for the
             remaining transitions */
18           let β ∈ work_set
19           τ_new := τ_old.β
20           work_set := work_set \ {β}
21           if β is a cutoff event then
22             cutoff_procs_new :=
               cutoff_procs ∪ {proc(β)}
23           endif
24           enqueue(τ_new, cutoff_procs_new)
25         endw
26         if α is a cutoff event then
27           cutoff_procs :=
             cutoff_procs ∪ {proc(α)}
28         endif
29       endif
30     endw
31     dequeue()
32   endw
33 end
```

7. EXPLOITING PREDICATE STRUCTURE

Recall that every ideal of $(E, \rightarrow)$ corresponds to a state of $States(\sigma_E)$. Thus, there can be as many as $|2^E|$ states in

$States(\sigma)$. In general, detecting temporal properties of the form $EF(\psi)$ on a trace σ_E is NP-complete in $|E|$. In this section, we present some special classes of predicates such that for a predicate ψ belonging to any of these classes, the temporal property $EF(\psi)$ can be detected on a trace σ_E in time that is **polynomial** in $|E|$. The notation $s \models \psi$ denotes that the state s satisfies the predicate ψ.

These predicate classes and their detection algorithms were first introduced by one of the authors in [9, 2], in the context of detecting predicates within a single poset. A novel contribution of *our* paper is to apply these detection algorithms, for the first time, to model-checking.

7.1 Linear predicates

It is well-known [4] that the set of all ideals of a poset forms a lattice under the subset relation $\subseteq$. In particular, if G and H are ideals of the poset $(E, \rightarrow)$, then $G \cap H$ and $G \cup H$ are also ideals of $(E, \rightarrow)$.

DEFINITION 7. *A predicate ψ is said to be* **meet-closed** *in a trace σ_E iff for every pair of ideals G and H of $(E, \rightarrow)$:*

$$(fin_{\sigma_G} \models \psi \wedge fin_{\sigma_H} \models \psi) \Rightarrow (fin_{\sigma_{G \cap H}} \models \psi)$$

Many useful predicates are meet-closed:

- A local variable is one whose scope is limited to exactly one process. A *local predicate* is a predicate that is defined using only local variables from a single process. It is easily verified that local predicates are meet-closed.
- If ψ_1 and ψ_2 are meet-closed, then so is $\psi_1 \wedge \psi_2$.

We define the *restriction* of an ideal G to a process P_i as:

$$G \mid_{P_i} \stackrel{def}{=} \{e | e \in G \wedge e \in P_i\}$$

DEFINITION 8. *Let G be an ideal of $(E, \rightarrow)$ such that $fin_{\sigma_G} \not\models \psi$. We say that process P_i is* **crucial** *at G if and only if, for all ideals H of $(E, \rightarrow)$:*

$$(G \subseteq H) \wedge (fin_{\sigma_H} \models \psi) \Rightarrow (G \mid_{P_i} \neq H \mid_{P_i})$$

Note that since $G \subseteq H$, $G \mid_{P_i} \neq H \mid_{P_i}$ is equivalent to saying that $G \mid_{P_i}$ is a strict subset of $H \mid_{P_i}$. Put simply, if the state fin_{σ_G} is reached during an execution of the program, the predicate ψ will not be satisfied in any state reached from G *unless* the crucial process executes at least one event.

Meet-closed predicates exhibit the following interesting property, which makes them particularly amenable to efficient detection algorithms.

THEOREM 5. *Let G be any ideal of the trace σ_E. Let ψ be a predicate such that $fin_{\sigma_G} \not\models \psi$. If ψ is meet-closed in σ_E, then there exists a crucial process at G.*

PROOF. We prove the contrapositive. Assume that there is no crucial process at G. Then, for every process P_i, there exists an ideal H_i such that $G \subseteq H_i$, and $H_i \mid_{P_i} = G \mid_{P_i}$, and $fin_{\sigma_{H_i}} \models \psi$. By Constraint 3, each event $e \in H_i$ belongs to exactly one process. Therefore, $\bigcap_i H_i = G$. Since, $fin_{\sigma_G} \not\models \psi$, this means that ψ is not meet-closed. $\square$

We can now devise an algorithm to detect $EF(\psi)$ within a trace $\sigma = [s_0, w]$, for a meet-closed predicate ψ. For an interleaving sequence w, let $w \mid_{P_i}$ be the sequence derived by erasing from w all the events not belonging to process P_i. For example, if $w = \alpha_1\beta_1\alpha_2\beta_2$, where $\alpha_1, \alpha_2 \in P_1$ and $\beta_1, \beta_2 \in P_2$, then $w \mid_{P_1} = \alpha_1\alpha_2$.

Algorithm 3 shows the procedure for detecting $EF(\psi)$ for meet-closed predicates ψ. In the algorithm, $next(w \mid_{P_i})$ returns the next event from process P_i in the sequence $w \mid_{P_i}$. If no such event is found, it returns $\emptyset$. The algorithm examines the current state fin_{σ_G}, corresponding to the ideal G, to see if the predicate is satisfied. If not, then it identifies the crucial process P_i, and picks the next event, e, from $w \mid_{P_i}$. Recall that any state that satisfies ψ must correspond to an ideal that includes e and is a strict superset of G. The smallest such ideal is $G \cup \downarrow e$. Therefore, the algorithm begins its next iteration with $G \cup \downarrow e$.

Algorithm 3: detect_EF

```
   input  : A trace σ = [s0, w], a meet-closed predicate ψ.
   output: true if EF(ψ) in σ, false otherwise.
1  begin
2  |  G := ∅ /* Initially, fin_σG = s0 */
3  |  while fin_σG ⊭ ψ do
4  |  |  P_i := crucial(G, ψ)
5  |  |  e := next(w |_Pi)
6  |  |  if e = ∅ then
7  |  |  |  /* no more events from P_i in w */
   |  |  |  return false /* ¬EF(ψ) */
8  |  |  else
9  |  |  |  G = G ∪ ↓ e /* The smallest ideal larger
   |  |  |  than G that contains e */
10 |  |  endif
11 |  endw
12 |  return true /* fin_σG ⊨ ψ */
13 end
```

Assuming it takes $O(C)$ time to find the crucial process at each step, it is clear that Algorithm 3 detects $EF(\psi)$ on a trace σ_E in $O(C.|E|)$ time.

DEFINITION 9. *A predicate ψ is* **linear** *in a trace σ_E iff:*

- *ψ is meet-closed in σ_E, and*
- *the crucial process can be identified in $O(|E|^k)$ time, for some constant $k \geq 0$.*

For a linear predicate, the time complexity of Algorithm 3 is $O(|E|^k)$ for some constant $k \geq 1$. As noted earlier, local predicates are meet-closed. Clearly, for a local predicate defined on local variables from process P_i, the crucial process is P_i. Thus, $EF(\psi)$ can be detected in $O(|E|)$ time for a local predicate ψ. Now, consider a predicate $\psi' = \psi_1 \wedge \psi_2 \wedge ... \wedge \psi_m$, where each ψ_j is a local predicate. Clearly, this predicate is meet-closed. Furthermore, if a state does not satisfy this predicate, then there exists a ψ_j that evaluates to false in that state. The failing conjunct can be identified in $O(m)$ time. Since each conjunct is a local predicate, this means that the crucial process can be identified in $O(m)$ time. Thus, the predicate ψ' is linear and $EF(\psi')$ can be detected in $O(m.|E|)$ time.

7.2 0-1 sum predicates

Another useful class of predicates are those of the form $x_1 + x_2 + ... + x_n > k$, where the x_i are local variables, and k is a constant. A special case is where each of the x_i can

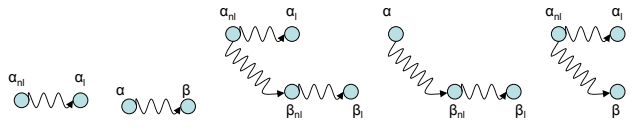

Figure 2: The relation $\rightsquigarrow$. (a) $\alpha \in E_{NL}$. (b)$\alpha, \beta \in E_L$, (c) $\alpha, \beta \in E_{NL}$, (d) $\alpha \in E_L, \beta \in E_{NL}$, and (e) $\alpha \in E_{NL}, \beta \in E_L$. Note that in (b) - (e), we also have $\alpha \rightarrow \beta$.

take only a value of either 0 or 1. We call such predicates **0-1 sum** predicates. 0-1 sum predicates can be used to detect mutual exclusion violation ($EF(\sum_i incs_i > 1)$). Or, to detect if there are more than k copies of a k-licensed software in use at once ($EF(\sum_i in\_use_i > k)$).

Let φ represent the 0-1 sum predicate $(x_1 + x_2 + ... + x_n > k)$. We reduce the problem of detecting $EF(\varphi)$ in a trace σ_E to the problem of computing the width of a poset[2].

An event $\alpha \in P_i$ is called a *local event* iff it affects only local variables at P_i (including the program counter at P_i). All other events are called *non-local events*. That is, the set of events E is partitioned into the set of local events, E_L, and the set of non-local events, E_{NL}. We now *split* each event $\beta \in E_{NL}$ into two sub-events, β_{nl} and β_l, where β_{nl} affects *only* non-local variables (including message channels), and β_l affects only local variables, including the program counter. Note that β_l is a local event, but is not a member of E_L. β can now be considered the sequential composition of $\beta_n l$ and β_l, *i.e.*, $\beta \equiv \beta_{nl}.\beta_l$. The splitting process transforms E_{NL} into a set of sub-events, $\hat{E}_{NL}$.

Let $\hat{E} = E_L \cup \hat{E}_{NL}$. We now transform the poset $(E, \rightarrow)$ into another poset $(\hat{E}, \rightsquigarrow)$, where the relation $\rightsquigarrow$ is the *smallest* transitive relation that satisfies each of the following (as shown in Figure 2):
(a) $\alpha \in E_{NL} \Rightarrow \alpha_{nl} \rightsquigarrow \alpha_l$
(b) $(\alpha, \beta \in E_L) \wedge (\alpha \rightarrow \beta) \Rightarrow \alpha \rightsquigarrow \beta$
(c) $(\alpha, \beta \in E_{NL}) \wedge (\alpha \rightarrow \beta) \Rightarrow \alpha_{nl} \rightsquigarrow \beta_{nl}$
(d) $(\alpha \in E_L) \wedge (\beta \in E_{NL}) \wedge (\alpha \rightarrow \beta) \Rightarrow \alpha \rightsquigarrow \beta_{nl}$
(e) $(\alpha \in E_{NL}) \wedge (\beta \in E_L) \wedge (\alpha \rightarrow \beta) \Rightarrow \alpha_{nl} \rightsquigarrow \beta$

For any $H \subseteq E$, let $\hat{H}$ denote the "expanded" set of events obtained by splitting the non-local events of H. That is,

$$\hat{H} \stackrel{def}{\equiv} \{\alpha | \alpha \in (H \cap E_L)\} \cup \{\alpha_{nl}, \alpha_l | \alpha \in (H \cap E_{NL})\}$$

Let $frontier(\hat{H})$ denote the set of maximal events, under $\rightsquigarrow$, from each process P_i in $\hat{H}$:

$$frontier(\hat{H}) \stackrel{def}{\equiv} \{\alpha | \alpha \in (P_i \cap \hat{H}) \wedge (\nexists \beta \in (P_i \cap \hat{H}) :: \alpha \rightsquigarrow \beta)\}$$

That is, $frontier(\hat{H})$ contains the "latest" event in $\hat{H}$ from each process P_i. From Figure 2, it is clear that $frontier(\hat{H})$ can contain only local events.

The proof for the following lemma is left to the reader (see [8, 22]).

LEMMA 5. *G is an ideal of $(E, \rightarrow)$ iff $\forall \alpha, \beta \in frontier(\hat{G}) : (\alpha \not\rightsquigarrow \beta) \wedge (\beta \not\rightsquigarrow \alpha)$.*

[2]The *width* of a poset is equal to the maximum number of mutually incomparable elements in the poset.

Let G be an ideal of $(E, \rightarrow)$ such that $fin_{\sigma_G} \models \varphi$. Let α_j be the (local) event from P_j in $frontier(\hat{G})$. Let $\ell(\alpha_j)$ denote the local state (*i.e.*, valuation of local variables) on P_j reached upon execution of α_j. Since $fin_{\sigma_G} \models \varphi$, there must exist a set Π of $(k+1)$ processes such that $\forall P_j \in \Pi$, $(x_j = 1)$ in the local state $\ell(\alpha_j)$. By Lemma 5, $\forall i, j \in \Pi : (\alpha_i \not\rightsquigarrow \alpha_j) \wedge (\alpha_j \not\rightsquigarrow \alpha_i)$.

Let $\mathcal{E} = \bigcup_i \{\alpha_i \in P_i | (x_i = 1) \text{ in } \ell(\alpha_i)\}$ That is, $\mathcal{E}$ is the set of all events that lead to a local state in which any x_i is set to 1. Thus, in order to detect $EF(\varphi)$, we simply need to determine whether the poset $(\mathcal{E}, \rightsquigarrow)$ has width greater than k, where $(\mathcal{E}, \rightsquigarrow)$ is the sub-poset of $(\hat{E}, \rightsquigarrow)$ induced by the relation $\rightsquigarrow$ on the set $\mathcal{E}$.

Tomlinson and Garg [23] presented an algorithm that solves this problem in $O(k.m.n(k + \log n))$ time, where $m = |\mathcal{E}|$ and n is the number of processes in the program.

8. EXPERIMENTAL RESULTS

For our implementation, we modified the popular model-checking tool, SPIN [12]. SPIN provides a choice of using either breadth-first search or depth-first search techniques to explore the state space. It uses a static reduction method [13] that is based on the partial order reduction techniques described in [18, 19], to generate a reduced state space graph. By static, we mean that the dependency relations are computed offline, during the parsing stage, before the model checking run is initiated. The reduced transition graph is constructed by picking a persistent set (based on the computed dependency relation) to explore at each state.

We refer to our modified version of SPIN as TC-SPIN (Trace Cover SPIN). We used SPIN's existing static reduction mechanism for computing dependency relations and choosing persistent sets. The algorithms we have described in this paper use a breadth-first search strategy, so our experimental comparisons were also done against SPIN's breadth-first search mechanism. Note that our algorithms can easily be adapted to a depth-first search strategy. The changes we made to SPIN are as follows:

- Each trace produced by Algorithm 2 is stored in an auxiliary data structure, eponymously named *trace*, which consists of an array of linked lists. Each element of the array corresponds to a process, and each linked list stores the sequence of local states corresponding to $w \mid_{P_i}$, for each interleaving sequence w created by Algorithm 2.
- We implemented the mechanism for assigning vector timestamps to each event executed. These vector timestamps were stored alongs side the local process states in the linked lists for each $w \mid_{P_i}$.
- In a traditional breadth-first search, new states are stored in a hashtable. When an executed transition leads to a state that already exists in the hashtable, the current path is not explored further. In TC-SPIN, our criteria for stopping the exploration of a path is based on cut-off events. Accordingly, we used SPIN's hashtable implementation, but modified it to store the state $fin_{\sigma_{\downarrow e}}$ for each event e that was explored by our algorithm. When a cut-off event is identified, we prune our search space accordingly, as described in Algorithm 2.
- We implemented the predicate detection algorithms presented in Section 7. The detection algorithms were invoked on each individual trace as soon as the construction of that trace was completed.

Protocol	Property	SPIN, no reduction			SPIN, P.O. reduction			TC-SPIN		
		Time (sec)	Memory (MB)	States	Time (sec)	Memory (MB)	States	Time (sec)	Memory (MB)	States
Dining philosophers	$EF(eating[i] \wedge eating[(i+1) mod N])$	***	***	***	759.71	439.116	2116120	0.03	1.253	83
Leader election	$EF(nr\_leaders > 1)$	***	***	***	777.24	64.741	238569	75.02	92.697	118971
Mutual exclusion	$EF(incs > 1)$	25.31	349.823	652365	2.51	26.239	46880	0.05	2.653	187

Table 1: Experimental results in the absence of errors in the verified protocols.

*** denotes "ran out of memory"

Protocol	Property	SPIN, no reduction			SPIN, P.O. reduction			TC-SPIN		
		Time (sec)	Memory (MB)	States	Time (sec)	Memory (MB)	States	Time (sec)	Memory (MB)	States
Dining philosophers	$EF(eating[i] \wedge eating[(i+1) mod N])$	41.86	257.049	1141680	10.22s	43.340	170619	0.03	1.253	81
Leader election	$EF(nr\_leaders > 1)$	***	***	***	547.41	44.773	159750	53.53	69.247	87435
Mutual exclusion	$EF(incs > 1)$	19.61	276.607	510828	1.59	15.385	26126	0.01	2.653	181

Table 2: Experimental results in the presence of safety violations in the verified protocols.

Tables 1 and 2 present our experimental results from the verification of the following three protocols:
1) Chandy and Misra's distributed dining philosophers protocol [1], with six philosophers ($N = 6$). We checked for the safety property that no pair of neighboring philosophers can ever eat simultaneously.
2) Dolev, Klawe and Rodeh's leader election protocol on a unidirectional ring [5] of six processes. We used random initialization to assign id's to the processes in the ring. The safety property to be verified was that there was never more than one leader in the ring.
3) Ricart and Agarwala's distributed mutual exclusion protocol [21] on five processes. The safety property to be checked was that there was at most one process in the critical section at any time.

The results in Table 1 are from verification runs where the protocols had no errors in them. For the second set of results, reported in Table 2, we introduced errors (safety violations) in each of the protocols, and measured the time and memory required to find these errors. In all our experiments, we compiled the verifier with -DBFS (for breadth-first search), and -DXUSAFE (to facilitate p.o. reduction). For SPIN, the safety properties were specified with an *assert* statement in a separate monitor process. For TC-SPIN runs, we specified our predicates in a different file. Consequently, for TC-SPIN runs only, we had to disable dataflow optimizations (-o1) during verifier generation because variables that were flagged as being "dead" were actually being read in our predicates file.

As seen from the results, partial order reduction helps tremendously in all three protocols. However, in SPIN, a persistent set can *only* consist of transitions that are invisible with respect to the property to be detected[3]. In TC-SPIN, a persistent set can consist of both visible and invisible transitions, that is, our reduction is insensitive to the visibility of transitions. It has been observed [10] that the effectiveness of partial order reduction deteriorates rapidly with an increase in the number of visible transitions. Therefore, the greater the percentage of visible transitions in a protocol, the greater the advantage of using TC-SPIN when compared to SPIN.

Another interesting observation is that TC-SPIN actually uses more memory than SPIN, for the same number of states stored. This is because TC-SPIN stores additional information for each event executed. In addition to storing the global state $fin_{\sigma_{\downarrow e}}$ corresponding to each event e, TC-SPIN also stores the vector timestamp of e, and the local state reached upon executing e. SPIN, on the other hand, only stores the global state reached upon executing an event. We are currently pursuing the implementation of some optimizations to reduce the memory consumed by TC-SPIN.

[3]This is not a limitation of SPIN itself, but rather of p.o. reduction techniques in general.

9. RELATED WORK

Our approach is a hybrid of an interleaving model such as is used in labeled transition graphs, and a true concurrency model, such as that used in Petri net unfoldings. In our model, we explore all interleavings of events that arise due to local non-deterministic choice within a process, while using a partial order representation to *avoid* exploring all interleavings of concurrent events. We differ from interleaving-based approaches as we avoid explicit generation of global states by using a partial order representation. Petri nets unfoldings also use a partial order representation of the state space. Unfoldings were originally used for deadlock detection by McMillan in [17]. Esparza later presented an algorithm for using unfoldings to verify safety properties expressed in a limited logic [6]. However, these algorithms were proposed for low-level Petri nets, which are inherently

unscalable. Low-level net representations tend to be quite large even for very simple high level programs, making them impractical for use in the verification of (large) real-world programs.

10. CONCLUDING REMARKS

In this paper, we proposed a new approach for model checking concurrent and distributed programs. Our approach exploits the compactness of a partial order representation to avoid state space explosion during model generation. Further, we exploit the *structure* of the predicate to verify safety properties in time that is polynomial in the size of the generated model. We presented such polynomial-time algorithms for two tractable classes of predicates - linear predicates, and 0-1 sum predicates. Our experimental results confirm that our proposed approach results in significant savings in the time and memory required for verification, compared to partial order reduction techniques. For example, we verified a leader election protocol in 75.02 seconds, whereas partial order reduction techniques verified the same protocol in 777.24 seconds.

The algorithms we presented in this paper were limited to the detection of safety properties of the form $EF(\varphi)$. A future direction of research is to develop efficient algorithms for the detection of properties involving additional temporal operators such as AF and EG. Another direction of research is to discover other tractable classes of predicates that allow for polynomial-time detection algorithms.

11. REFERENCES

[1] K. M. Chandy and J. Misra. The drinking philosophers problem. *ACM Trans. Program. Lang. Syst.*, 6(4):632–646, 1984.

[2] C. M. Chase and V. K. Garg. Efficient detection of restricted classes of global predicates. In *WDAG '95: Proceedings of the 9th International Workshop on Distributed Algorithms*, pages 303–317, London, UK, 1995. Springer-Verlag.

[3] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite state concurrent system using temporal logic specifications: a practical approach. In *POPL '83: Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 117–126, New York, NY, USA, 1983. ACM Press.

[4] B. Davey and H. Priestly. *Introduction to Lattices and Order*. Cambridge University Press, Cambridge, 1990.

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

[6] J. Esparza. Model checking using net unfoldings. In *TAPSOFT '93: Selected papers of the colloquium on Formal approaches of software engineering*, pages 151–195, Amsterdam, The Netherlands, The Netherlands, 1994. Elsevier Science Publishers B. V.

[7] C. Fidge. Logical time in distributed computing systems. *Computer*, 24(8):25–33, Aug. 1991.

[8] F.Mattern. Virtual time and global states of distributed systems. In *Proc. of the International Workshop on Distributed Algorithms*, pages 215–226, 1989.

[9] V. K. Garg and B. Waldecker. Detection of weak unstable predicates in distributed programs. *IEEE Trans. Parallel Distrib. Syst.*, 5(3):299–307, 1994.

[10] R. Gerth, R. Kuiper, D. Peled, and W. Penczek. A partial order approach to branching time logic model checking. In *Proceedings of the Third Israel Symposium on the Theory of Computing and Systems (ISTCS'95), Tel Aviv, Israel, January 4-6, 1995*, 1995.

[11] P. Godefroid. *Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem*, volume 1032. Springer-Verlag Inc., New York, NY, USA, 1996.

[12] G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, Sept. 2003.

[13] G. J. Holzmann and D. Peled. An improvement in formal verification. In *Proceedings of the 7th IFIP WG6.1 International Conference on Formal Description Techniques VII*, pages 197–211. Chapman & Hall, Ltd., 1995.

[14] M. Z. Kwiatkowska. Event fairness and non-interleaving concurrency. *Formal Aspects of Computing*, 1:213–228, 1989.

[15] L. Lamport. Time, clock and the ordering of events in a distributed system. *Communications of the ACM (CACM)*, 21(7):558–565, July 1978.

[16] A. W. Mazurkiewicz. Basic notions of trace theory. In *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, School/Workshop*, pages 285–363, London, UK, 1989. Springer-Verlag.

[17] K. L. McMillan. Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits. In *CAV '92: Proceedings of the Fourth International Workshop on Computer Aided Verification*, pages 164–177, London, UK, 1993. Springer-Verlag.

[18] D. Peled. *All from One, One for All: on Model Checking Using Representatives*, volume 697 of *Lecture Notes in Computer Science*. Springer-Verlag, 1993.

[19] D. Peled. Combining partial order reductions with on-the-fly model-checking. In *CAV '94: Proceedings of the 6th International Conference on Computer Aided Verification*, pages 377–390, London, UK, 1994. Springer-Verlag.

[20] D. Peled and T. Wilke. Stutter-invariant temporal properties are expressible without the next-time operator. *Inf. Process. Lett.*, 63(5):243–246, 1997.

[21] G. Ricart and A. K. Agrawala. An optimal algorithm for mutual exclusion in computer networks. *Commun. ACM*, 24(1):9–17, 1981.

[22] F. B. Schneider and L. Lamport. Paradigms for distributed programs. In *Distributed Systems: Methods and Tools for Specification, An Advanced Course, April 3-12, 1984 and April 16-25, 1985 Munich*, pages 431–480, London, UK, 1985. Springer-Verlag.

[23] A. I. Tomlinson and V. K. Garg. Monitoring functions on global states of distributed programs. *J. Parallel Distrib. Comput.*, 41(2):173–189, 1997.

[24] A. Valmari. *Stubborn Sets for Reduced State Space Generation*, volume 483 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, Germany, 1990.