A compact, predicate-independent state space representation for model checking

Sujatha Kashyap
Dept. of Electrical and Computer Engineering
University of Texas at Austin
Austin, TX 78712
Email: kashyap@ece.utexas.edu

***Abstract*— State space explosion is a significant obstacle in the formal verification of properties on distributed programs. Much effort has been directed in the area of state space reduction for model checking distributed programs. Such research efforts mainly employ one of two techniques: (a) construction of a *reduced* explicit state space, or (b) a compact representation of the state space.**

In this paper, we explore the tradeoffs between these approaches, and the limitations of current model-checking techniques. We propose a new approach that uses a compact state space representation, while at the same time facilitating efficient detection of properties in the model.

Our representation uses a hybrid of an interleaving approach and a true concurrency representation. We provide experimental results comparing the performance of our implementation against a wide variety of popular model-checking tools.

I. Introduction

The fundamental obstacle in the use of model checking for the formal verification of concurrent and distributed systems is the problem of *state space explosion*. A distributed program is modeled by a set of concurrently executing processes, each of which executes a set of *local* events. The behavior of such a system is modeled by the set of all *global* states that could be reached during any execution of the program, also known as the set of *consistent global states* of the distributed program. The number of consistent global states of a distributed program is, in general, exponential in the total number of local process states. Thus, distributed programs are particularly prone to state space explosion.

A significant portion of the recent research in the area of model checking has focused on tackling the problem of state space explosion. There are two orthogonal approaches to tackling the state space explosion problem. The first approach is to construct a *reduced* state space, by omitting the exploration of some interleavings of events. *Partial order reduction techniques* [1], [2], [3], [4] are a particularly successful example of this approach.

Partial order reduction techniques exploit the fact that exploring *all* possible interleavings of concurrent events is not always necessary for the verification of a given property. In particular, they show that, under certain conditions, considering a single *representative* interleaving is sufficient for the detection of a subclass of linear temporal logic (LTL) formulae. For this reason, the use of partial order techniques is also called *model checking using representatives*. In particular, they consider formulae that are expressible in the subset of the logic LTL without the use of the next-time operator X. This subset of LTL is denoted by LTL_{-X}. In precise terms, given an LTL_{-X} property φ, partial order techniques explore only a subset of the global state space that is provably sufficient to check the property φ.

Partial order techniques have been incorporated into several existing verification tools, such as the popular model checking tool, SPIN [5]. Unfortunately, in such approaches, the reduction is performed with respect to the properties to be verified. Thus, the amount of reduction achieved in the size of the state space very much depends on the properties to be verified [6].

Another popular approach is to represent the entire state space in a compact notation. Net unfoldings of Petri Nets [7] are an example of this approach. Net unfoldings preserve the concurrency and causality relations between events on different processes, without explicitly constructing the entire state space. However, verifying properties on such representations is a P-SPACE complete problem. Net unfoldings are often called "true concurrency" approaches.

While an interleaving approach may result in an exponentially large representation of the state space, properties on the constructed state space can be verified in time that is polynomial to the size of the represen-

tation. On the other hand, true concurrency approaches encode the state space compactly, but the problem of verifying properties on these representations is P-SPACE complete.

Our approach uses a combination of the interleaving and true concurrency models to represent the complete state space graph of the system to be verified. Our representation maintains the concurrency and causality relations in the program, while allowing for the efficient verification of properties.

The ideas presented in this paper capitalize on a large body of existing research done in the area of *predicate detection* in distributed computations ($e.g.$, [8], [9], [10], [11], [12], [13]). Our proposed approach avoids using an interleaving representation for concurrent events wherever possible, choosing instead to use a true concurrency (partial order) representation for sets of concurrent events. In particular, we use a hybrid of a modified Hasse diagram representation [14], combined with the labeled transition graph representation traditionally used in model checking. Our representation can be viewed as a labeled transition graph where each node in the graph is a partially ordered set. Thus, each node in the graph can potentially encode an exponential number of global states without explicitly enumerating these states. We call this hybrid the *poset-LTS* representation.

For experimental purposes, we have modified the popular model checking tool SPIN to use our algorithms. Thus, we can verify properties on program specifications written in PROMELA.

This paper is organized as follows. In Section II, we present an overview of the currently used partial order reduction techniques, also known as model checking using representatives. We also highlight the limitations of these techniques. In Section III, we present the model of distributed programs that our approach is based on, and present the relationship between partial order representations and distributed programs. This will lay the groundwork for our proposed representation, which is presented in Section IV. In Section V, we discuss how model checking can be performed using our proposed representation. Section VI presents some experimental results and compares the performance of our implementation against several popular model checking tools, including DSSZ , SMV, PEP and SPIN. Finally, we present our conclusions and directions for future work in Section VII.

II. Model checking using representatives

The partial order reduction techniques discussed in [2], [3], [4] construct a reduced state space graph through the use of a modified depth-first search (or breadth-first search) algorithm. This modified depth-first search technique is also used in SPIN for implementing partial order reductions [15], [5].

A *labeled transition graph* is a quintuple $G = (V, S, S_0, E, \Gamma)$, where [16]:

- V is a finite set of variables (state holding elements).
- $S = 2^V$ is the set of all possible valuations for the variables in V. S is called the *state set*.
- $S_0 \subseteq S$ is the set of *initial states*.
- E is the set of possible *events* or *actions* in the system.
- $\Gamma \subseteq S \times E \times S$ is the *transition relation*, with the restriction that $((s, \alpha, t) \in \Gamma \wedge (s, \alpha, t') \in \Gamma) \Rightarrow (t = t')$. For convenience of notation, if $(s, \alpha, t) \in \Gamma$, we say $t = \alpha(s)$.

In the rest of this paper, we only consider systems with a *designated initial state*. In traditional (non-symbolic) model checking using depth-first search or breadth-first search, the state transition graph is generated by exploring *all* the events enabled at a given state $s \in S$. We denote the set of all events enabled in s by $enabled(s)$. Partial order reduction techniques, on the other hand, explore only a subset of the events in $enabled(s)$, and thus construct a *subgraph* of G, given by $G' = (V, S', S_0, E, \Gamma')$, where $S' \subseteq S$ and $\Gamma' \subseteq \Gamma$. The subset of events in $enabled(s)$ chosen for exploration is called an *ample set* and is denoted by $ample(s)$. The choice of $ample(s)$ at a given state s depends on the property, φ, to be checked. It has been shown ([2], [3]) that, by imposing certain constraints on the set $ample(s)$, the property φ holds in G' iff it holds in G.

Partial order reduction techniques use the notions of *independence* and *invisibility* of events [6], which we discuss in the next subsection. We also discuss the conditions that $ample(s)$ must satisfy in order to maintain the correctness of the reduction. This discussion is taken from [6].

A. Independence and Invisibility

An independence relation $I \subseteq E \times E$ is a symmetric, antireflexive relation that satisfies the following two conditions for each $s \in S$ and each $(\alpha, \beta) \in I$:

- *Enabledness*: If $\alpha, \beta \in enabled(s)$, then $\alpha \in enabled(\beta(s))$.

- *Commutativity*: If $\alpha, \beta \in enabled(s)$, then $\alpha(\beta(s)) = \beta(\alpha(s))$.

Note that the definition above uses the symmetry of I. The enabledness condition states that the execution of α does not disable the event β. Thus, β must remain enabled in the state reached upon execution of α. The commutativity condition states that executing independent transitions in either order results in the same state. The notion of independence is illustrated in Figure 1. Events that are not independent are said to be dependent on each other.

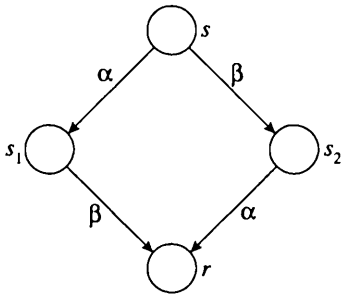

Fig. 1. Execution of independent transitions.

It is easy to see that, in order for the two interleavings of α and β in Figure 1 to be replaced by a single interleaving in the reduced graph, the property to be checked must not be sensitive to the choice of the states s_1 and s_2. That is, the property must hold in s_1 iff it holds in s_2. This leads us to the definition of *invisibility*.

Let $L : S \rightarrow 2^{AP}$ be a labeling function that labels each state $s \in S$ with a set of atomic propositions. An event α is invisible with respect to a set of atomic propositions $AP' \subseteq AP$ iff, for each pair of states $s, s' \in S$ such that $s' = \alpha(s)$, $L(s) \cap AP' = L(s') \cap AP'$. That is, an event α is invisible iff its execution from *any* state does not change the values of the variables in AP'. An event that is not invisible is said to be *visible*.

We now state the conditions that $ample(s)$ must satisfy in order to preserve the correctness of partial order reduction. For brevity, we simply state the conditions without explanation. The interested reader is referred to [6] for details.

- **C0:** $ample(s) = \Phi$ iff $enabled(s) = \Phi$.
- **C1:** Along every path in the full state graph that starts at s, the following condition holds: a transition that is dependent on a transition in $ample(s)$ cannot be executed without a transition in $ample(s)$ being executed first.
- **C2:** If $ample(s) \neq enabled(s)$, then every event $\alpha \in ample(s)$ is invisible.
- **C3:** A cycle is not allowed if it contains a state in which some transition α is enabled, but α is never included in $ample(s)$ for any state s on the cycle.

The following lemma from [6] is a direct consequence of condition **C1**:

Lemma 1: [6] The transitions in $enabled(s) \backslash ample(s)$ are all independent of those in $ample(s)$.

B. Limitations of model checking using representatives

The invisibility condition in **C2** poses a significant restriction on the construction of ample sets. We demonstrate this in the following trivial example.

Example 1: Consider n processes $P_1, P_2, ..., P_n$. Each process P_i contains a local variable x_i whose value it changes independently of all other processes. Each process P_i goes through exactly m local states, $< s_{i,0}, s_{i,1}, ..., s_{i,m} >$, and in each local state, it executes an event that changes the value of x_i. Initially, $x_i = 0$ for all $i \in \{1, ..., n\}$. In the traditional method of construction of the labeled transition graph using depth-first search, there would be m^n reachable states in the constructed graph. Suppose we are interested in checking a local property, such as $x_3 = k$. Using partial order reduction techniques as described above, it suffices to explore an interleaving in which, starting from the initial state, P_1 first goes through its m states, followed by P_2 going through its m states, and so on, until P_n goes through its m states. Thus, the number of explored states in this case would only be $O(mn)$.

However, if we needed to check a global property, such as $x_1 + x_2 + ... + x_n \geq k$, then partial order techniques are ineffective because of the invisibility condition **C2**. In fact, in this example, we would be forced to choose $ample(s) = enabled(s)$ for each state, starting from the initial state, since each event *does* change the value of the propositional variables in the property to be checked. Thus, each event is *visible* with respect to the property to be checked. In this case, the number of states explored using partial order reduction techniques would remain m^n, that is, the reduced graph would be the same as the original labeled transition graph. In fact, the overhead of futilely trying to construct an ample set would only incur additional overheads compared to the naive approach of constructing the labeled transition graph by exploring all possible interleavings!

In the next section, we describe the model of the distributed programs that our proposed representation

operates on, in particular, the specific assumptions that we have made about distributed programs. We also discuss relevant background work.

III. Distributed Programs and Partially Ordered Sets

We model a distributed program as a set of processes $\{P_1, ..., P_n\}$ with no shared memory and no global clock. Each process P_i starts at an initial local state and executes a set of local events sequentially. We model each such process P_i as a set of local states s_{i1}, ..., s_{im}. Each process P_i has a set of local variables X_i. The value of a local variable on a process changes only upon transitions between local states on that process. Thus, in any state s_{ij} on process P_i, the value of each variable $x \in X_i$ is well-defined.

We assume that we are given a per-process labeled transition graph, which can be constructed by traditional depth-first search techniques. Thus, for each process P_i, we have the labeled transition graph $G_i = (X_i, S_i, s_{i,0}, E_i, \Gamma_i)$, where:

- X_i is the set of local variables of P_i,
- S_i is the local state set of P_i,
- $s_{i,0} \in S_i$ is the designated initial local state of P_i,
- E_i is the set of possible events that can be executed by P_i, and
- $\Gamma_i \subseteq S_i \times E_i \times S_i$ is the local transition relation for P_i, with the restriction that $((s, \alpha, t) \in \Gamma_i \wedge (s, \alpha, t') \in \Gamma_i) \Rightarrow (t = t')$. For convenience of notation, if $(s, \alpha, t) \in \Gamma_i$, we say $t = \alpha(s)$.

Currently, our technique only handles cases where the process transition diagrams have no cycles (including self-loops on a state), *i.e.*, processes that cannot go into an infinite loop on any input.

Process P_i communicates with process P_j through asynchronous messages along the communication channel C_{ij}. Communication channels are not assumed to be FIFO, but they are assumed to be reliable. When P_i sends a message to P_j, a $send(j, msg)$ event is executed on P_i, and a $receive(i, msg)$ event is executed at P_j when the message is received.

A single execution of a distributed program is called a *computation*. Lamport [17] observed that a distributed computation can be viewed as partially-ordered set of states, ordered under what he called the *happened-before relation*, denoted by $\rightarrow$. Formally, this happened-before relation states that, given two states s and t from the set of all states $S = \bigcup_{i=1}^{n} S_i$, $s \rightarrow t$ iff s occurs before t on the same process, or a message is sent from state s and received at state t, or $\exists u \in S : (s \rightarrow u) \wedge (u \rightarrow t)$.

C. Fidge [18] and F. Mattern [19] independently proposed a timestamping mechanism that can be employed during the execution of a distributed computation to capture the happened-before relation of that computation. Under this timestamping mechanism, each process maintains a *vector clock*, and associates a timestamp to each of its local states. The *vector clock* is a vector of n numbers, where n is the total number of processes in the system. The *vector timestamp* of state s is denoted by $s.v$, and satisfies the condition:

$$\forall s, t \in S : s \rightarrow t \Leftrightarrow s.v < t.v$$

where the $<$ operation performs a component-wise comparison of the two vector clocks, as follows:

$$s.v < t.v \Leftrightarrow (\forall i \in \{1, ..., n\} : s.v[i] \leq t.v[j]) \wedge (\exists j \in \{1, ..., n\} : s.v[j] < t.v[j])$$

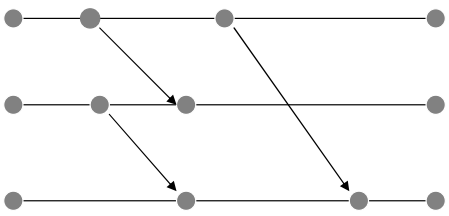

Fig. 2. Modified Hasse diagram representation of a distributed computation.

Thus, a distributed computation can be represented by a Hasse diagram [14], under the $\rightarrow$ relation. However, a more convenient (and intuitive) notation is to represent the computation in the form of a chain-decomposed poset, consisting of n chains, where chain $\mathcal{C}_i$ contains the local states of process P_i. Figure 2 depicts this representation.

Note that, since each process executes sequentially, the local states of process P_i in a given distributed computation are totally ordered under the $\rightarrow$ relation. Furthermore, send and receive events induce a partial ordering on the set of all states, S.

Thus, we define a distributed computation as a set of states together with the happened-before relation, and denote it by $(S, \rightarrow)$. States $s, t \in S$ that are incomparable under the $\rightarrow$ relation are said to be *concurrent*, denoted by $s||t$. Formally, $(s||t) \Leftrightarrow (s \not\rightarrow t \wedge t \not\rightarrow s)$.

The poset representation of a distributed computation implicitly encodes all the *feasible* interleavings of events that could have occurred during the computation via the incomparability of states. That is, if $s||t$, then in an

interleaving representation, both $< s, t >$ and $< t, s >$ are valid interleavings.

The reachability (global state space) graph of a distributed computation must only contain global states that *could* have occurred during the execution of the program, that is, only global states that can be reached while respecting the happened-before relation. We call such states *consistent global states*. It is easy to see that a consistent global state corresponds to a *down-set* or *order-ideal* of the poset $(S, \rightarrow)$. Formally, a consistent global state $\mathcal{H}$ of a distributed computation $(S, \rightarrow)$ is a set of states that satisfies the following condition:

$$(s \in \mathcal{H} \wedge t \rightarrow s) \Rightarrow (t \in \mathcal{H})$$

It has been shown in [20], [19], [12] that the set of all consistent global states of a finite distributed computation forms a finite distributive lattice under the $\rightarrow$ relation. Each path from the initial global state to the final global state of this finite distributive lattice represents a possible interleaving of events that could have occurred during the execution of the program [8].

It is well-known that the number of ideals of a poset can be exponential in the size of the poset itself [14], [21]. The poset representation of a single execution of a distributed program thus pithily encodes an exponential number of global states. However, in order to formally verify the correctness of the program, we need to consider *all* possible executions of the program. The program itself may be viewed as the set of all its possible executions.

In the following section, we propose a new *poset-LTS* representation that can be used to represent the global state space of a distributed *program*, that is, the set of all its possible computations.

IV. POSET-LTS REPRESENTATION

Given a total ordering for all the local events that occurred on each process during the execution of a distributed program, the happened-before relation captures all the consistent global states of the program under that particular total ordering of local events. Note that, in addition to the total ordering of local events, the only other order imposed by the happened-before relation is between send and receive events. The order imposed between send and receive events is a "true" order, since a receive event cannot execute until the corresponding send has executed. However, the ordering on local events is not necessarily a true order, as the following example illustrates.

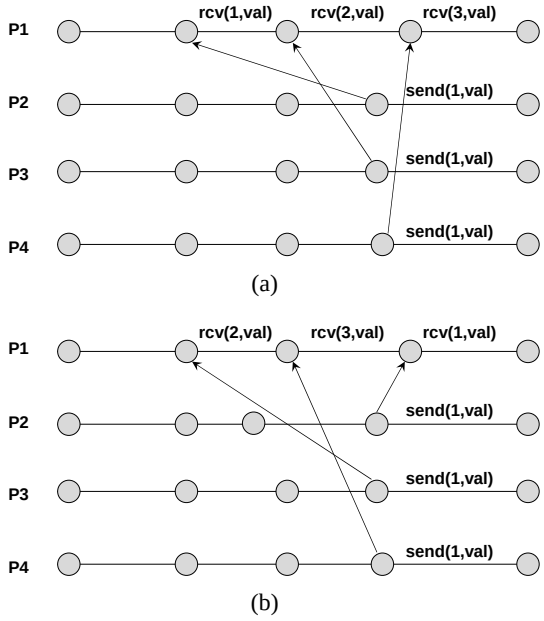

Fig. 3. Two executions of the program in Example 2.

Example 2: Consider a distributed computation on n processes, $\{P_1, P_2, ...P_n\}$, where each of the processes $P_2...P_n$ performs a local computation, and finally sends the result of its local computation to P_1. The job of P_1 is simply to accumulate the results sent to it by all other processes. Figure 3(a) and (b) show two possible executions of the program for four processes. Since message transfers are asynchronous, there can be $3!$ possible interleavings of the receive events at P_1, depending on the order in which the messages arrive.

Thus, its initial state, P_1 has $n-1$ events enabled, namely, the receive events corresponding to each of the processes $P_2, ..., P_n$. However, a single happened-before diagram can only represent one interleaving of these events.

From the above example it is clear that, if a process in a distributed program has more than one event enabled at any local state, then a single happened-before diagram cannot capture all the consistent global states of the program. The *poset-LTS* representation we propose here aims at capturing all the global states of the program.

Before we present our approach, however, it would be useful to revisit Example 1 to demonstrate the advantages of using a poset representation, as opposed to an interleaving representation. This is done in the following example.

Example 3: We consider the scenario presented in Example 1. The process transition graphs for this example are presented in Figure 4(a). Since each process

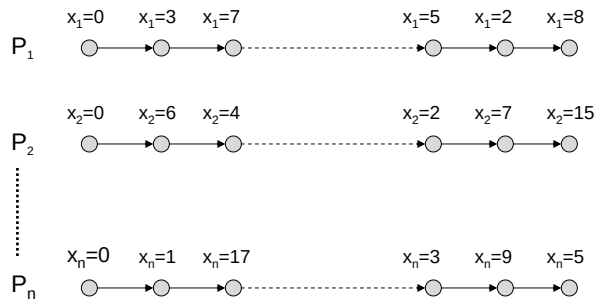

Fig. 4. Process transition graphs for Example 3

has exactly one event enabled at each state, a single happened-before diagram captures all the possible global states of this program. In fact, Figure 4(a) also doubles as the poset representation for the program, since there are no message transfers in this program, $i.e.$, we do not need to impose any ordering between events on different processes.

Thus, the poset representation encodes the global state space of the program in $O(mn)$ space. We now consider the problem of checking the property $x_1+x_2+...+x_n \geq k$. Let us call this property φ. Here, we call upon the large body of previous work done in the area of predicate detection in distributed computations. In particular, we can use the algorithm by Chase and Garg in [9] to detect $EF(\varphi)$. The algorithm runs in $O(m^2n^2 \log mn)$ time and $O(mn)$ space. Thus, we have achieved an exponential reduction in the time and space required for verifying this example program, compared to the $O(m^n)$ space and time bounds for model checking using representatives.

To capture the state space of a distributed program in which more than one event may be enabled at a local state, we use a hybrid of the interleaving representation and the poset representation. We call this hybrid the *poset-LTS* representation. Under this representation, the global state space of the program is represented by a transition graph in which each node of the graph is a poset. Thus, each node of the graph may itself encode an exponential number of states, and the graph as a whole encodes *all* the reachable states of the program. In particular, each path from the initial node to a final node in the graph corresponds to an execution of the distributed program.

The construction of a node (poset) in the graph proceeds as follows. As long as there exactly one local event possible at any process, we "grow" the node by executing that single local event on the process (if the event is enabled), and add the resulting local state to the node's

struct *node* $\{$
$structstateQueue[1..n]$: **type** array of queues;
$struct\ node\ *pred$: **type** set;
$\}$

Fig. 5. Structure of a node.

struct *stateQueue* $\{$
$localState$: **type** int;
$chanContents$: **type** char *;
$vc[1..n]$: **type** array of int;
$\}$

Fig. 6. Structure of a state queue.

poset representation. The data structure representing a node is shown in Figure 5. The node maintains a queue for each process, to which we add the local states reached upon executing local events on that process. The state of message channels is also stored along with the local state, in the state queue. The queue corresponding to process P_i is represented by $stateQueue[i]$. The last entry in the queue for a process P_i in node Q is given by $lastinQueue(Q.stateQueue[i])$, and represents the latest state reached by P_i in node Q.

Each process maintains a vector clock, which is updated according to the rules in [19]. The vector clock of each local state is stored in the state queue, as shown in Figure 6.

A process *blocks* when there is more than one local event possible at its current local state. For a node Q, the set $Q.blocked$ contains all the blocked processes. A node stops growing when all processes are either blocked or have no enabled events in the current state. The algorithm for growing a node is shown in Procedure 1. The procedure $predDetect()$ uses Garg and Waldecker's algorithm for detection of weak conjunctive predicates [11] to detect if the global state has already been visited. The global state given by S is just a conjunction of the local states (the state of the message channels is also maintained as part of the local state of a process), hence can be detected as a conjunction of local predicates by the algorithm in [11].

Let $S_0 = \bigcup_{i\in\{1,...,n\}}\{s_i\}$, where $s_i \in P_i$, be the initial local states of node Q. When the node has stopped growing, the final global state S is given by

$$S = \bigcup_{i\in\{1,...,n\}} \{lastinQueue(Q.stateQueue[i])\}$$

When a process has more than one local event enabled, this corresponds to a non-deterministic choice in the

Procedure 1 growNode(node Q)

```
1: /* Processes that have at least one event enabled in their current state */
2: enabledProcs = {P_i | enabled(lastinQueue(Q.stateQueue[i])) ≥ 1}
3: while enabledProcs \ Q.blocked ≠ Φ do
4:    /* some process can make progress */
5:    let P_i ∈ enabledProcs \ Q.blocked
6:    /* P_i is not blocked, and has at least one event enabled */
7:    s = lastinQueue(Q.stateQueue[i]) /* current local state of P_i */
8:    if |possible(s)| > 1 then
9:       Q.blocked = Q.blocked ∪ {P_i}
10:   else if |possible(s)| == 1 then
11:      /* possible(s) has exactly one event, execute it */
12:      let α ∈ possible(s)
13:      if α ∈ enabled(s) then
14:         s' = α(s)
15:         addToStateQueue(s', Q.stateQueue[i])
16:         S = ⋃_{i∈{1,...,n}} {lastinQueue(Q.stateQueue[i]}
17:         if predicateDetect(S) == true then
18:            /* the new global state has already been explored */
19:            return Q
20:         end if
21:      end if
22:   end if
23:   enabledProcs = {P_i | enabled(lastinQueue(Q.stateQueue[i])) ≥ 1}
24: end while
25: return Q
```

program. Thus, at this point, we switch to an interleaving representation and explore each enabled transition from S.

To construct the complete *poset-LTS* representation of a program, we start with a designated initial state, and create a new node with its initial global state set to the designated initial state. We then call upon procedure $poset\_BFS()$ to create the rest of the state space using a breadth-first search. The relevant algorithms are shown in Procedures 2 and 3. Depth-first search could also be used instead of breadth-first search.

The procedure $createNode(S)$ deserves special mention. Rather than creating a new node Q with the state queues initialized to the local states in S, we chose to create the new node Q with its state queues initialized to the local states in S', given by:

$$\forall_{i\in\{1,..,n\}} : S'[i] = \forall_{j\in\{1,..,n\}} min(S[j].vc[i])$$

In other words, for each process P_i, the local state $S'[i]$ is given by the least local state in the i^{th} component of the the vector clocks of the local states in S.

The reason for this is as follows. Consider a consistent global state G such that $\exists i \in \{1,..,n\} : S[i] \preceq G[i]$. From the properties of vector clocks, the least consistent global state that includes $G[i]$ is given by the vector clock of $G[i]$ [19]. Since $G[i] \preceq S[i]$, the following must hold:

$$\forall j \in \{1,..,n\} : S'[j] \preceq G[j]$$

As a consequence of the above observation, Q contains any consistent global state that is made up of local states that occur either in Q or any of its predecessor nodes.

V. Model checking under the poset-LTS representation

Efficient algorithms for the detection of $EF(\varphi)$ have been proposed for predicates φ of the following forms, among others:

- Conjunctions of local predicates [11]
- Linear and regular predicates [9], [12]
- Bounded sum predicates (predicates of the form $x_1 + ... + x_m \leq k$) [9]

Procedure 2 poset_BFS(node currNode)

```
1: enQueue(currNode)
2: while (Q = headOfQueue())! = null do
3:    growNode(Q)
4:    S = ⋃_{i∈{1,...,n}} {lastinQueue(Q.stateQueue[i]}
5:    workSet(S) = enabled(S)
6:    while workSet(S) ≠ Φ do
7:       let β ∈ workSet(S)
8:       workSet(S) = workSet(S) − {β}
9:       S' = β(S)
10:      if predicateDetect(S') == false then
11:         /* this is a new state */
12:         Q_new = createNode(S')
13:         enQueue(Q_new)
14:      end if
15:   end while
16:   /* done with this node, remove from the queue */
17:   deQueue()
18: end while
```

Procedure 3 main

```
1: /* S_0 is the initial state */
2: S_0 = ⋃_{i∈{1,..,n}} s_{i,0}
3: Q_0 = createNode(S_0)
4: poset_BFS(Q_0)
```

These algorithms operate efficiently on the partial order representation of a distributed computation, and are particularly well-suited to model-checking $EF(\varphi)$ on the *poset-LTS* representation.

The $predicateDetect()$ routine used in our algorithms (Figures 2, 1) uses the algorithm in [11] for detection $EF(\varphi)$ for conjunctions of local predicates. We can easily detect $EF(\varphi)$ for linear, regular and bounded-sum predicates in our model, too. To perform this check on the fly, we would simply need to introduce a call to the appropriate predicate detection algorithm immediately preceding the call to $predicateDetect()$ in Figures 2 and 1.

VI. Experimental Results: Leader Election Protocol

We have implemented an extension to SPIN that takes a PROMELA language specification and builds the corresponding *poset-LTS* representation of the state space graph. Our implementation currently has the ability to detect $EF(\varphi)$ for linear predicates, regular predicates and bounded sum predicates.

Using Dekker's two-process mutual exclusion algorithm as our sample program to be verified, we compared the efficiency of our implementation against a wide variety of popular model-checking tools. The property to be detected was the violation of safety, that is, to check whether both processes could be in the critical section at the same time ($EF : incs1 \wedge incs2$).

Table VI lists the model checking tools we compared, the corresponding description language used to specify the mutual exclusion protocol for each tool, and the time taken to verify the program in each case. Of all the description languages listed in Table VI, PROMELA provides the most diverse array of constructs for the specification of distributed protocols. This observation contributed to our decision to use PROMELA as the description language for our implementation.

Since our implementation is a modification of SPIN, it is useful to compare the performance of our implementation against that of SPIN. Figure 7 lists the sample Promela program that we used to compare the performance of our approach against the partial order reduction techniques used in SPIN.

```
#define N 5
proctype node(byte number; chan in, out){
 byte num;
  xr in;  xs out;
  out!number;
  in?num;
}
init{
 byte proc;
chan q[N]= [N] of { byte };
 proc = 1;
 do
 :: proc <= N ->
        run node(proc, q[proc-1], q[proc%N]);
        proc++
 :: proc > N ->
        break
 od
}
```

Fig. 7. Sample PROMELA program for comparing performance against SPIN.

Model Checker	Description Language	Time (sec)
DSSZ	BPN2	0.07
DSSZ	CFA	0.07
DSSZ	Senil	0.07
SMV	APNN	0.01
SMV	BPN2	0.36
SMV	CFA	0.01
SMV	Senil	0.03
PEP	APNN	0.01
PEP	CFA	0.02
PEP	Senil	0.01
SPIN	PROMELA	0.01
poset-LTS	PROMELA	0.01

TABLE I
VERIFYING SAFETY OF DEKKER'S MUTUAL EXCLUSION ALGORITHM

In SPIN, safety properties are specified by placing an *assert*statement in a separate process. In order to allow the assert statement to have access to the process-local variable *num* in *proctype node*, we defined *num* to be a global variable for the version of the program that was passed through the SPIN model checker. The assertion statement was specified as follows:

```
active proctype safe()
{assert (num != 0)}
```

The program in Figure 7 was passed unchanged to our implementation of poset-LTS. The predicate to be detected in our case was: $P_0.number = 0 \wedge P_1.number = 1 \wedge P2_number = 2$.

Table VI shows the time taken to verify the program when the number of processes was varied from 5 to 50. At 20 processes, the SPIN model checker was still verifying the program after 580 seconds, and was killed at that point.

VII. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented a new approach for representing the state space of distributed programs. Our proposed approach has significant advantages compared to existing techniques such as model checking using representatives. In particular, we have shown that our approach is independent of the properties to be verified on the program. Our approach preserves the set of all reachable states of the program, whereas model checking using representatives produces a reduced state graph which may not contain all the reachable states of the program.

Similar to Petri Net unfoldings, the $poset - LTS$ representation described in this paper encodes concurrency and causality information among processes. However, a significant difference is that our representation maintains this information in a way that facilitates efficient detect of a number of classes of predicates, including bounded sum predicates. We are not aware of any unfolding-based

Number processes	SPIN Time (sec)	poset-LTS Time (sec)	SPIN states stored	poset-LTS states stored
5	0.01	0.01	160	29
10	0.08	0.01	7199	54
15	9.11	0.01	311342	79
20	*	0.01	*	104
25	*	0.02	*	129
50	*	0.55	*	254

TABLE II

COMPARING SPIN PERFORMANCE AGAINST POSET-LTS.

method that can detect bounded sum predicates in time that is polynomial in the size of the unfolding, even for occurrence nets. Furthermore, PROMELA, which is the input specification language for our implementation, provides a much richer programming environment than the Petri Net specification languages used in unfolding-based techniques.

A significant limitation of our approach is that we are currently limited to the detection of predicates of the form $EF(\varphi)$ for a limited class of predicates φ. A future area of research is to extend our approach to additional classes of predicates. Currently, our implementation can check for conjunctions of local predicates. We are currently working on implementing the detection of bounded sum predicates into our SPIN-based tool.

REFERENCES

[1] W. T. Overman, "Verification of concurrent systems: function and timing," Ph.D. dissertation, Univ. of California, 1981.

[2] A. Valmari, *Stubborn Sets for Reduced State Space Generation*, ser. Lecture Notes in Computer Science, G. Rozenberg, Ed. Berlin, Germany: Springer-Verlag, 1990, vol. 483.

[3] D. Peled, *All from One, One for All: on Model Checking Using Representatives*, ser. Lecture Notes in Computer Science, C. Courcoubetis, Ed. Springer-Verlag, 1993, vol. 697.

[4] P. Godefroid and P. Wolper, "A partial approach to model checking," in *Papers presented at the IEEE symposium on Logic in computer science*. Academic Press, Inc., 1994, pp. 305–326.

[5] 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*. Chapman & Hall, Ltd., 1995, pp. 197–211.

[6] E. M. Clarke, O. Grumberg, and D. Peled, *Model Checking*. Cambridge, MA: MIT Press, 2000, ch. 10.

[7] J. Esparza, "Model checking using net unfoldings." *Lecture Notes in Computer Science: Proc. TAPSOFT 93*, vol. 668, pp. 613–628, 1993.

[8] R. Cooper and K. Marzullo, "Consistent detection of global predicates," in *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, Santa Cruz, California, 1991, pp. 163–173.

[9] C. Chase and V. K. Garg, "On techniques and their limitations for the global predicate detection problem," in *Proceedings of the Workshop on Distributed Algorithms*, Le Mont-Saint-Michel, France, Sept. 1995, pp. 303–307.

[10] B. Charron-Bost *et al.*, "Local and temporal predicates in distributed systems," *ACM Transactions on Programming Languages and Systems*, vol. 17, no. 1, pp. 157–179, Jan. 1995.

[11] V. K. Garg and B. Waldecker, "Detection of weak unstable predicates in distributed programs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, no. 3, pp. 299–307, Mar. 1994.

[12] V. K. Garg and N. Mittal, "On slicing a distributed computation," in *21st International Conference on Distributed Computing Systems (ICDCS 01)*, Washington-Brussels-Tokyo, Apr. 2001, pp. 322–329.

[13] A. Sen and V. K. Garg, "Detecting temporal logic predicates in the happened before model," in *International Parallel and Distributed Processing Symposium (IPDPS)*, Florida, Apr. 2002. [Online]. Available: http://www.computer.org/proceedings/ipdps/1573/symposium/

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

[15] G. J. Holzmann, D. Peled, and M. Yannakakis, *On nested depth-first search*. American Mathematical Society, 1996.

[16] R. Alur, R. K. Brayton, T. A. Henzinger, S. Qadeer, and S. K. Rajamani, "Partial-order reduction in symbolic state space exploration," in *Proceedings of the 9th International Conference on Computer Aided Verification*. Springer-Verlag, 1997, pp. 340–351.

[17] L. Lamport, "Time, clock and the ordering of events in a distributed system," *Communications of the ACM (CACM)*, vol. 21, no. 7, pp. 558–565, July 1978.

[18] C. Fidge, "Logical time in distributed computing systems," *Computer*, vol. 24, no. 8, pp. 25–33, Aug. 1991.

[19] F.Mattern, "Virtual time and global states of distributed systems," in *Proceedings of the International Workshop on Distributed Algorithms (WDAG)*, 1989, pp. 215–226.

[20] D. B. Johnson and W. Zwaenepoel, "Recovery in distributed systems using optimistic message logging and check-pointing," *J. Algorithms*, vol. 11, no. 3, pp. 462–491, 1990.

[21] J. S. Provan and M. O. Ball, "The complexity of counting cuts and of computing the probability that a graph is connected," *SIAM Journal on Computing*, vol. 12, pp. 777–788, 1983.