Dynamic Partial-Order Reduction for Model Checking Software

Cormac Flanagan
University of California at Santa Cruz
cormac@cs.ucsc.edu

Patrice Godefroid
Bell Laboratories, Lucent Technologies
god@bell-labs.com

ABSTRACT

We present a new approach to partial-order reduction for model checking software. This approach is based on initially exploring an arbitrary interleaving of the various concurrent processes/threads, and *dynamically* tracking interactions between these to identify backtracking points where alternative paths in the state space need to be explored. We present examples of multi-threaded programs where our new dynamic partial-order reduction technique significantly reduces the search space, even though traditional partial-order algorithms are helpless.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs

General Terms

Algorithms, Verification, Reliability

Keywords

Partial-order reduction, software model checking

1. INTRODUCTION

Over the last few years, we have seen the birth of the first *software model checkers* for programming languages such as C, C++ and Java. Roughly speaking, two broad approaches have emerged. The first approach consists of automatically extracting a model out of a software application by statically analyzing its code and abstracting away details, applying traditional model checking to analyze this abstract model, and then mapping abstract counter-examples back to the code or refining the abstraction (e.g., [1, 14, 4]). The second approach consists of systematically exploring the state space of a concurrent software system by driving its executions via a run-time scheduler (e.g., [10, 27, 5]). Both of these approaches to software model checking have their advantages and limitations (e.g., [11]).

POPL'05, January 12–14, 2005, Long Beach, California, USA.

In the context of the second approach, partial-order reduction seems (so far) to be the most effective technique for reducing the size of the state space of concurrent software systems at the implementation level. Two main *core* partial-order reduction techniques are usually considered: persistent/stubborn sets and sleep sets.

In a nutshell, the *persistent/stubborn set* technique [25, 9] computes a provably-sufficient subset of the set of enabled transitions in each visited state such that unselected enabled transitions are guaranteed not to interfere with the execution of those being selected. The selected subset is called a persistent set, while the most advanced algorithms for computing such sets are based on the notion of stubborn sets [25, 9]. These algorithms exploit information about "which operations on which communication objects each process might execute in the *future*". This information is typically obtained from a static analysis of the code. Minimally, such a static analysis can simply attempt to identify objects accessible by a single process only, and then classify operations on such objects as local (e.g., [5]).

In contrast, the *sleep set* technique (see [9]) exploits information on dependencies exclusively among the transitions enabled in the current state, as well as information recorded about the *past* of the search. Both techniques can be used simultaneously and are complementary [9].

In the presence of cycles in the state space, these techniques must be combined with additional conditions to make sure the transition selection is fair with respect to all processes in order to verify properties more elaborate than deadlock detection, such as checking arbitrary safety and liveness properties. For instance, an *ample set* (see [2]) is a persistent set that satisfies additional conditions sufficient for LTL model checking. Techniques for dealing with cycles are mostly orthogonal to the two "core" techniques mentioned above, which are sufficient for detecting deadlocks.

In what follows, we will assume that the state spaces we consider do not contain any cycles, and focus the discussion on detecting deadlocks and safety-property violations such as assertion failures (e.g., specified with `assert()` in C). Note that acyclic state spaces are quite common in the context of model checking of software implementations: the execution of most software applications eventually terminates, either because the application is input driven and reacts to external events specified in a test driver encoding only finite sequences of inputs, or because the execution length is bounded at run/test time and hence forced to terminate.

Unfortunately, existing persistent/stubborn set techniques suffer from a severe fundamental limitation: in the context

Figure 1: Indexer Program.

```
Thread-global (shared) variables:
    const int size = 128;
    const int max = 4;
    int[size] table;

Thread-local variables:
    int m = 0, w, h;

Code for thread tid:
    while (true) {
        w := getmsg();
        h := hash(w);
        while (cas(table[h],0,w) == false) {
            h := (h+1) % size;
        }
    }

    int getmsg() {
        if (m < max ) {
            return (++m) * 11 + tid;
        } else {
            exit(); // terminate
        }
    }
    int hash(int w) {
        return (w * 7) % size;
    }
```

of concurrent software systems executing arbitrary C, C++ or Java code, determining "which operations on which communication objects each process might execute in the future" with acceptable precision is often difficult or impossible to compute precisely. If this information is too imprecise, persistent/stubborn sets techniques cannot prune the state space very effectively. Sleep sets can still be used, but used alone, they can only reduce the number of explored transitions, not the number of explored states [9], and hence cannot avoid state explosion.

To illustrate the nature of this problem, consider the program `Indexer` shown in Figure 1, where multiple concurrent threads manipulate a shared hash table. Each thread has a a thread identifier `tid` $\in \{1, \ldots, n\}$ and receives a number of incoming messages `w` and inserts each message into the hash table at corresponding index `h=hash(w)`. If a hash table collision occurs, the next free entry in the table is used. All hash table entries are initially 0. The atomic compare-and-swap instruction `cas(table[h],0,w)` checks if `table[h]` is initially 0; if so, then it updates `table[h]` to `w` and returns `true`, and otherwise returns `false` (without changing `table[h]`).

A static *alias* analysis for determining when two different threads can access the same memory location would need to know all the possible messages received by all the threads as well as predict the hash values computed for all such messages, for all execution paths. Since this is clearly not realistic, static analyses conservatively assume that every access to the hash table may access the same entry. The latter is equivalent to treating the entire hash table as a single shared variable, and prevents partial-order reduction techniques from significantly pruning the state space of this program. Instead, all possible interleavings of accesses to the hash table are still explored, resulting in state explosion and making model checking intractable for all but a small number of threads.

We propose in this paper a new approach to partial-order reduction that avoids the inherent imprecisions of static alias analyses. Our new algorithm starts by executing the program until completion, resolving nondeterminism completely arbitrarily, and it *dynamically* collects information about how threads have communicated during this specific execution trace, such as which shared memory locations were read or written by which threads and in what order. This data is then analyzed to add backtracking points along the trace that identify alternative transitions that need to be explored because they might lead to other execution traces that are not "equivalent" to the current one (i.e., are not linearizations of the same partial-order execution). The procedure is repeated until all alternative executions have been explored and no new backtracking points need be added. When the search stops, all deadlocks and assertion failures of the system are guaranteed to have been detected.

For the `Indexer` example, if it is detected dynamically during the first execution trace of the program that the various threads access disjoint memory location, then no backtracking points are added along that trace, and the reduced state space with our dynamic partial-order reduction is a single path. This turns out to be the case for this program with up to 11 threads, as we will show in Section 5.

The paper is organized as follows. After some background definitions, we present in Section 3 a general dynamic partial-order reduction algorithm for detecting deadlocks in acyclic state spaces. In section 4, we discuss how to optimize this algorithm to the case of multithreaded programs. In Section 5, we present preliminary experimental results on some small examples. Section 6 discusses other related work, and we conclude with Section 7.

2. BACKGROUND DEFINITIONS

2.1 Concurrent Software Systems

We consider a concurrent system composed of a finite set $\mathcal{P}$ of *threads* or *processes*, and define its state space using a *dynamic semantics* in the style of [10]. Each process executes a sequence of *operations* described in a deterministic sequential program written in a language such as C, C++ or Java. The processes communicate by performing atomic operations on communication objects, such as shared variables, semaphores, locks, and FIFO buffers. In what follows, processes that share the same heap are called *threads*. Threads are thus a particular type of processes. Unless otherwise specified, the algorithms discussed in this paper apply to both processes and threads.

Operations on communication objects are called *visible operations*, while other operations are *invisible*. The execution of an operation is said to *block* if it cannot currently be completed; for instance, an operation "acquire(l)" may block until the lock is released by an operation "release(l)". We assume that only executions of visible operations may block.

A *state* of the concurrent system consists of the local state *LocalState* of each process, and of the shared state

SharedState of all communication objects:

$$\begin{array}{rcl} \mathit{State} & = & \mathit{SharedState} \times \mathit{LocalStates} \\ \mathit{LocalStates} & = & \mathcal{P} \rightarrow \mathit{LocalState} \end{array}$$

For $ls \in \mathit{LocalStates}$, we write $ls[p := l]$ to denote the map that is identical to ls except that it maps p to local state l.

A *transition* moves the system from one state to a subsequent state, by performing one visible operation of a chosen process, followed by a *finite* sequence of invisible operations of the same process, ending just before the next visible operation of that process. The transition $t_{p,l}$ of process p for local state $l \in \mathit{LocalState}$ is defined via a partial function:

$$t_{p,l} : \mathit{SharedState} \rightharpoonup \mathit{LocalState} \times \mathit{SharedState}$$

Let $\mathcal{T}$ denote the set of all transitions of the system. A transition $t_{p,l} \in \mathcal{T}$ is *enabled* in a state $s = \langle g, ls \rangle$ (where $g \in \mathit{SharedState}$ and $ls \in \mathit{LocalStates}$) if $l = ls(p)$ and $t_{p,l}(g)$ is defined. If t is enabled in s and $t_{p,l}(g) = \langle g', l' \rangle$, then we say the execution of t from s produces a unique[1] successor state $s' = \langle g', ls[p := l'] \rangle$, written $s \xrightarrow{t} s'$. We write $s \stackrel{w}{\Rightarrow} s'$ to mean that the execution of the finite sequence $w \in \mathcal{T}^*$ leads from s to s'.

We define the behavior of the concurrent system as a transition system $A_G = (\mathit{State}, \Delta, s_0)$, where $\Delta \subseteq \mathit{State} \times \mathit{State}$ is the *transition relation* defined by

$$(s, s') \in \Delta \text{ iff } \exists t \in \mathcal{T} : s \xrightarrow{t} s'$$

and s_0 is the initial state of the system.

In any given state $s = \langle g, ls \rangle$, let $\mathit{next}(s,p) = t_{p,ls(p)}$ denote the (unique) next transition to be executed by process p. For any transition $t_{p,l}$, let $\mathit{proc}(t_{p,l}) = p$ denote the process executing the transition (we thus assume all processes have disjoint sets of transitions). A state in which no transition is enabled is called a *deadlock*, or a *terminating state*.

The state transformation resulting from the execution of a transition may vary depending on the current state. For instance, if the next visible operation of thread p in a state s is `read(x)` in the program:

```
{if (read(x)) then i=0 else i=2}; write(x);
```

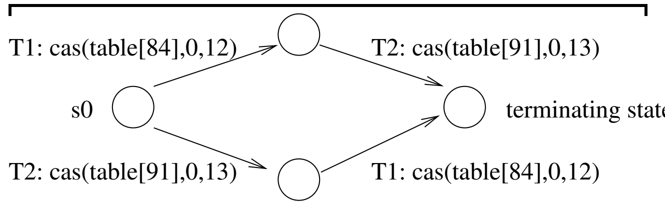
where `x` is a shared variable and `i` is a local variable, the invisible operation(s) following `read(x)` will depend on the value of `x`. However, the transition $t = \mathit{next}(s,p)$ is still unique and can thus be viewed as the entire block between the braces. Note that $\mathit{next}(s,p)$ does not change even if other processes execute other transitions changing the value of `x` from s: for all s' such that $s \stackrel{w}{\Rightarrow} s'$ where w does not contain any transition from p, we have $\mathit{next}(s',p) = \mathit{next}(s,p)$.

Consider again the `Indexer` example of Figure 1. Its state space A_G (for two threads and `max`=1) is shown in Figure 2. Transitions in A_G are labeled with the visible operation of the corresponding thread transition being executed. Nondeterminism (branching) in A_G is caused only by concurrency. This state space contains a single terminating state (where both threads are blocked on their `exit()` statement) since the two threads access distinct hash table entries.

Observe how the above definition of state space collapses purely-local computations into single transitions, by combining invisible operations with the last visible one. This definition avoids including the (unnecessary) interleavings of invisible operations as part of the state space (hence already reducing state explosion), while still being provably sufficient for detecting deadlocks and assertion violations as shown in [10]. Also, a model checker for exploring such state spaces needs only control and observe the execution of visible operations, as is done in the tool VeriSoft [10, 11].

[1]To simplify the presentation, we do not consider operations that are nondeterministic [10] or that create dynamically new processes, although both features are compatible with the algorithms and techniques discussed in the paper.

Figure 2: State space for Indexer example for two threads $T1$ and $T2$ and `max`=1.



2.2 Definitions for Partial-Order Reduction

We briefly recall some basic principles of partial-order reduction methods. The basic observation exploited by these techniques is that A_G typically contains many paths that correspond simply to different execution orders of the same uninteracting transitions. When concurrent transitions are "independent", meaning that their execution does not interfere with each other, changing their order of execution will not modify their combined effect. This notion of *independency* between transitions and its complementary notion, the notion of dependency, can be formalized by the following definition (adapted from [15]).

Definition 1. *Let $\mathcal{T}$ be the set of transitions of a concurrent system and $D \subseteq \mathcal{T} \times \mathcal{T}$ be a binary, reflexive, and symmetric relation. The relation D is a* valid dependency relation *for the system iff for all $t_1, t_2 \in \mathcal{T}$, $(t_1, t_2) \notin D$ (t_1 and t_2 are* independent*) implies that the two following properties hold for all states s in the state space A_G of the system:*

1. *if t_1 is enabled in s and $s \xrightarrow{t_1} s'$, then t_2 is enabled in s iff t_2 is enabled in s'; and*
2. *if t_1 and t_2 are enabled in s, then there is a unique state s' such that $s \stackrel{t_1 t_2}{\Rightarrow} s'$ and $s \stackrel{t_2 t_1}{\Rightarrow} s'$.*

Thus, independent transitions can neither disable nor enable each other, and enabled independent transitions commute. This definition characterizes the properties of possible "valid" dependency relations for the transitions of a given system. In practice, it is possible to give easily-checkable conditions that are *sufficient* for transitions to be independent (see [9]). Dependency can arise between transitions of different processes that perform visible operations on the same shared object. For instance, two acquire operations on the same lock are dependent, and so are two write operations on the same variable; in contrast, two read operations on the same variable are independent, and so are two write or compare-and-swap operations on different variables (such as `cas(table[84],0,12)` and `cas(table[91],0,13)` in the program of Figure 1). To simplify the presentation, we assume in this paper that the dependency relation is not conditional [15, 9] and that all the transitions of a particular process are dependent.

Traditional partial-order algorithms operate as classical state-space searches except that, at each state s reached during the search, they compute a subset T of the set of transitions enabled at s, and explore only the transitions in T. Such a search is called a *selective search* and may explore only a subset of A_G. Two main techniques for computing such sets T have been proposed in the literature: the persistent/stubborn set and sleep set techniques. The first technique actually corresponds to a whole family of algorithms [25, 12, 13, 20], which can be shown to compute persistent sets [9]. Intuitively, a subset T of the set of transitions enabled in a state s of A_G is called *persistent in s* if whatever one does from s, while remaining outside of T, does not interact with T. Formally, we have the following [12].

DEFINITION 2. *A set $T \subseteq \mathcal{T}$ of transitions enabled in a state s is* persistent in s *iff, for all nonempty sequences of transitions*

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

from s in A_G and including only transitions $t_i \notin T$, $1 \leq i \leq n$, t_n is independent with all the transitions in T.

It is beyond the scope of this paper to review stubborn-set-like algorithms for computing persistent sets. In a nutshell, these algorithms can exploit information on the *static* structure of the system being verified, such as "from its current local state, process x could perform operation y on shared object z in some of its executions", and inferred (approximated) by a static analysis of the system code. For instance, see [9] for several such algorithms and a comparison of their complexity.

The new partial-order reduction algorithm introduced in the next section also explores at each visited state s the transitions in a persistent set in s. But unlike previously known algorithms, these persistent sets are computed *dynamically*.

Before presenting this new algorithm, we briefly recall some properties of persistent sets that will be used later. First, a selective search of A_G using persistent sets is guaranteed to visit all the deadlocks in A_G (see Theorem 4.3 in [9]). Moreover, if A_G is acyclic, a selective search using persistent sets is also guaranteed to visit all the reachable local states of every process in the system (see Theorem 6.14 in [9]), and hence can be used to detect violations of any property reducible to local state reachability, including violations of local assertions and of safety properties.

3. DYNAMIC PARTIAL-ORDER REDUCTION

We present in this section a new partial-order reduction algorithm that dynamically tracks interactions between processes and then exploits this information to identify backtracking points where alternative paths in the state space A_G need to be explored. The algorithm is based on a traditional depth-first search in the (reduced) state space of the system.

The algorithm maintains the traditional depth-first search stack as a transition sequence executed from the initial state s_0 of A_G. Specifically, a *transition sequence* $S \in \mathcal{T}^*$ is a (finite) sequence of transitions $t_1 t_2 \ldots t_n$ where there exist states $s_1, \ldots, s_{n+1}$ such that s_1 is the initial state s_0 and

$$s_1 \xrightarrow{t_1} s_2 \ldots \xrightarrow{t_n} s_{n+1}$$

Given a transition sequence S, we use the following notation:

- S_i refers to transition t_i;
- $S.t$ denotes extending S with an additional transition t;
- $dom(S)$ means the set $\{1, \ldots, n\}$;
- $pre(S, i)$ for $i \in dom(S)$ refers to state s_i; and
- $last(S)$ refers to s_{n+1}.

A transition $t \in \mathcal{T}$ can appear multiple times in a transition sequence S. We write $t_i = t_j$ to denote that transitions t_i and t_j are occurrences of a same transition in $\mathcal{T}$.

We say a transition t_1 *may be co-enabled with* a transition t_2 if there may exist some state in which both t_1 and t_2 are both enabled. For example, an acquire and release on the same lock are never co-enabled, but two write operations on the same variable may be co-enabled.

If two adjacent transitions in a transition sequence are independent, then they can be swapped without changing the overall behavior of the transition sequence. A transition sequence thus represents an equivalence class of similar sequences that can be obtained by swapping adjacent independent transitions. To help reason about the equivalence class represented by a particular transition sequence, we maintain a "happens-before" ordering relation on these transitions. The *happens-before* relation $\rightarrow_S$ for a transition sequence $S = t_1 \ldots t_n$ is the smallest relation on $\{1, \ldots, n\}$ such that

1. if $i \leq j$ and S_i is dependent with S_j then $i \rightarrow_S j$;
2. $\rightarrow_S$ is transitively closed.

By construction, the happens-before relation $\rightarrow_S$ is a partial-order relation, often called a "Mazurkiewicz's trace" [18, 9], and the sequence of transitions in S is one of the linearizations of this partial order. Other linearizations of this partial order yield "equivalent" transition sequences that can be obtained by swapping adjacent independent transitions.

We also use a variant of the happens-before relation to identify backtracking points. Specifically, the relation

$$i \rightarrow_S p$$

holds for $i \in dom(S)$ and process p if either

1. $proc(S_i) = p$ or
2. there exists $k \in \{i+1, \ldots, n\}$ such that $i \rightarrow_S k$ and $proc(S_k) = p$.

Intuitively, if $i \rightarrow_S p$, then the next transition of process p from the state $last(S)$ is not the next transition of process p in the state right before transition S_i in either this transition sequence or in any equivalent sequence obtained by swapping adjacent independent transitions.

The new partial-order reduction algorithm is presented in Figure 3. In addition to maintaining the current transition sequence or search stack S, each state s in the stack S is also associated with a "backtracking set", denoted $backtrack(s)$, which represents processes with a transition enabled in s that needs to be explored from s.

Whenever a new state s is reached during the search, the procedure Explore is called with the stack S with which the state is reached. Initially (line 0), the procedure Explore is called with the empty stack as argument. In line 2, $last(S)$

Figure 3: Dynamic Partial-Order Reduction Algorithm.

0 Initially: Explore($\emptyset$);

1 Explore(S) {
2   let $s = last(S)$;
3   for all processes p {
4   if $\exists i = max(\{i \in dom(S) \mid S_i$ is dependent and may be co-enabled with $next(s,p)$ and $i \not\rightarrow_S p\})$ {
5   let $E = \{q \in enabled(pre(S,i)) \mid q = p$ or $\exists j \in dom(S) : j > i$ and $q = proc(S_j)$ and $j \rightarrow_S p\}$;
6   if ($E \neq \emptyset$) then add any $q \in E$ to $backtrack(pre(S,i))$;
7   else add all $q \in enabled(pre(S,i))$ to $backtrack(pre(S,i))$;
8   }
9   }
10   if ($\exists p \in enabled(s)$) {
11         $backtrack(s) := \{p\}$;
12   let $done = \emptyset$;
13   while ($\exists\ p \in (backtrack(s) \setminus done)$) {
14   add p to $done$;
15   Explore($S.next(s,p)$);
16   }
17   }
18 }

is the state reached by executing S from the initial state s_0. Then, for all processes p, the next transition $next(s,p)$ of each process p in state s is considered (line 3). For each such transition $next(s,p)$ (which may be enabled or disabled in s), one then computes (line 4) the last transition i in S such that S_i and $next(s,p)$ are dependent (cf. Definition 1) and may be co-enabled, and such that $i \not\rightarrow_S p$.

If there exists such a transition i, there might be a race condition or dependency between i and $next(s,p)$, and hence we might need to introduce a "backtracking point" in the state $pre(S,i)$, i.e., in the state just before executing the transition i.[2] This is determined in line 5 by computing the set E of processes q with an enabled transition in $pre(S,i)$ that "happens-before" $next(s,p)$ in the current partial order $\rightarrow_S$. Intuitively, if E is nonempty, the execution of all the processes in E is necessary (although perhaps not sufficient) to reach transition $next(s,p)$ and to make it enable in the current partial order $\rightarrow_S$; in that case, it is therefore sufficient to add any single one of the processes in E to the backtracking set associated with $pre(S,i)$. In contrast, if E is empty, the algorithm was not able to identify a process whose execution is necessary for $next(s,p)$ to become enabled from $pre(S,i)$; by default (line 7), the algorithm then adds all enabled processes in $backtrack(pre(S,i))$.

Once the computation of possibly new backtracking points of lines 3–9 is completed, the search can proceed from the current state s. If there are enabled processes in s (line 10), any one of those is selected to be explored by being added to the backtracking set of s (line 11). As long as there are enabled processes in the backtracking set associated with the current state s that have not been explored yet, those processes will be executed one by one by the code of lines 13–15. When all the processes in $backtrack(s)$ have been explored this way, the search from s is over and state s is said to be "backtracked".

Note that the algorithm of Figure 3 is *stateless* [10]: it does not store previously visited states in memory since efficiently computing a canonical representation for states of large concurrent (possibly distributed) software applications is problematic and prohibitively expensive. Backtracking can be performed without storing visited states in memory, for instance by re-executing the program from its initial state, or "forking" new processes at each backtracking point, or storing only backtracking states using checkpointing techniques, or a combination of these [11].

To illustrate the recursive nature of our dynamic partial order reduction algorithm, consider two concurrent processes p_1 and p_2 sharing two variables `x` and `y`, and executing the two programs:

p_1: `x=1; x=2;`
p_2: `y=1; x=3;`

Assume the first (arbitrary and maximal) execution of this concurrent program is:

p_1`:x=1;` p_1`:x=2;` p_2`:y=1;` p_2`:x=3;`

Before executing the last transition p_2`:x=3` of process p_2, the algorithm will add a backtracking point for process p_2 just before the *last* transition of process p_1 that is dependent with it (the transition p_1`:x=2`), forcing the subsequent exploration of:

p_1`:x=1;` p_2`:y=1;` p_2`:x=3;` p_1`:x=2;`

Similarly, before executing the transition p_2`:x=3` in that second sequence, the algorithm will add a backtracking point for process p_2 just before p_1`:x=1`, which in turn will force the exploration of:

p_2`:y=1;` p_2`:x=3;` p_1`:x=1;` p_1`:x=2;`

[2] Only the last transition i in S satisfying the constraints of line 4 needs be considered: if there are other such transitions $j < i$ before i in S that require adding other backtracking points, these are added later through the recursion of the algorithm.

Note that the two possible terminating states (deadlocks) and three possible partial-order executions of this concurrent program are eventually explored. This example illustrates why it is sufficient to consider only the last transition in S in line 4 of our algorithm of Figure 3.

The correctness of the above algorithm is established via the following theorem.

THEOREM 1. *Whenever a state s is backtracked during the search performed by the algorithm of Figure 3 in an acyclic state space, the set T of transitions that have been explored from s is a persistent set in s.*

Proof: See Appendix. ■

Since the algorithm of Figure 3 explores a persistent set in every visited state, it is guaranteed to detect every deadlock and safety-property violation in any acyclic state space (see Section 2). The complexity of the algorithm depends on how the happens-before relation $\rightarrow_S$ is implemented and is discussed further in the next section.

Theorem 1 specifies the type of reduction performed by our new algorithm, as well as its complementarity and compatibility with other partial-order reduction techniques. In particular, any algorithm for computing statically persistent sets, such as stubborn-set-like algorithms, can be used in conjunction of the algorithm in Figure 3: in lines 5 and 7, replace $enabled(pre(S,i))$ by $PersistentSet(pre(S,i)))$, and in line 10, replace $enabled(s)$ by $PersistentSet(s)$, where the function $PersistentSet(s)$ computes "statically" a persistent set T in state s and returns $\{proc(t) \mid t \in T\}$. These modifications will restrict the search space to transitions contained in the statically-computed persistent sets for each visited state s, while using dynamic partial-order reduction to further refine these statically-computed persistent sets.

Moreover, sleep sets can also be used in conjunction with the new dynamic technique, combined or not with statically-computed persistent sets. In our context (i.e., for acyclic state spaces), sleep sets can be added exactly as described in [10]. The known benefits and limitations of sleep sets compared to persistent sets remain unchanged: used alone, they can only reduce the number of explored transitions, but used in conjunction with (dynamic or static) persistent set techniques, they can further reduce the number of states as well [9].[3]

We conclude this section by briefly discussing two optimizations of the algorithm of Figure 3.

1. Since any process q in E can be added to the set $backtrack(pre(S,i))$ in line 6, it is clearly preferable to pick a process q that is already in $backtrack(pre(S,i))$, whenever possible, in order to minimize the size of $backtrack(pre(S,i))$.

2. A more subtle optimization consists of not adding *all* enabled processes to $backtrack(pre(S,i))$ in line 7 when E is empty, but instead of selecting a single other process q enabled in $pre(S,i)$ and not previously executed from $pre(S,i)$, and to re-start a new persistent-set computation in $pre(S,i)$ with q as the initial process. However, to avoid circularity in this reasoning and to ensure the correctness of this variant algorithm, it is then necessary to "mark" process $proc(t_i)$ in state $pre(S,i)$ so that, if $proc(t_i)$ is ever selected to be backtracked in $pre(S,i)$ during this new persistent-set computation starting with q, yet another fresh persistent-set computation may be needed in $pre(S,i)$, and so on.

[3]There is a nice complementarity between sleep sets and our dynamic partial-order reduction algorithm: when a process q is introduced in $backtrack(pre(S,i))$ in line 6 or 7 because of a potential conflict between i and $next(s,p)$, there is no point in executing S_i following $next(pre(S,i),q)$ before $next(s,p)$ is executed; this optimization is captured exactly by sleep sets.

4. IMPLEMENTATION

In this section, we discuss how to implement the previous general algorithm. We assume that the system has m processes $p_1, \ldots, p_m$; that d is the maximum size of the search stack; and that r is the number of transitions explored in the reduced search space.

4.1 Clock Vectors

The implementation of the algorithm of Figure 3 is mostly straightforward, apart from identifying the necessary backtracking points in lines 3–9, which requires deciding the happens-before relation $i \rightarrow_S p$. A natural representation strategy for the happens-before relation is to use clock vectors [17]. A *clock vector* is a map from process identifiers to indices in the current transition sequence S:

$$CV \ = \ \mathcal{P} \rightarrow \mathcal{N}$$

We maintain a clock vector $C(p) \in CV$ for each process p. If process p_i has clock vector $C(p_i) = \langle c_1, \ldots, c_m \rangle$, then c_j is the index of the last transition by process p_j such that $c_j \rightarrow_S p_i$. More generally:

$$i \rightarrow_S p \quad \text{if and only if} \quad i \leq C(p)(proc(S_i))$$

Thus clock vectors allow us to decide the happens-before relation $i \rightarrow_S p$ in constant time.

We use $max(\cdot,\cdot)$ to denote the pointwise maximum of two clock vectors; $C[p_i := c'_i]$ to update the clock vector C so that the clock for process p_i is c'_i; and $\perp$ to denote the minimal clock vector:

$$\begin{aligned} max(\langle c_1,..,c_m \rangle, \langle c'_1,..,c'_m \rangle) &= \langle max(c_1,c'_1),..,max(c_m,c'_m) \rangle \\ \langle c_1,\ldots,c_m \rangle[p_i := c'_i] &= \langle c_1,\ldots,c_{i-1},c'_i,c_{i+1},\ldots,c_m \rangle \\ \perp &= \langle 0,\ldots,0 \rangle \end{aligned}$$

Whenever we explore a transition of a process p, we need to update the clock vector $C(p)$ to be the maximum of the clock vectors of all preceding dependent transitions. For this purpose, we also keep a clock vector $C(i)$ for each index i in the current transition sequence S.

Clock vectors can also be used to compute the set E as in line 5 of Figure 3. However, this requires $O(m^2.d)$ time per explored transition. Instead, for simplicity, our modified algorithm just backtracks on all enabled processes in the case where p is not enabled in $pre(S,i)$ (line 7). Note that this last modification is independent of the use of clock vectors.

Figure 4 presents a modified algorithm that maintains and uses these per-process and per-transition clock vectors. The code at lines 14.1–14.5 that updates these clock vectors requires $O(m.d)$ time per explored transition. Line 4 of the algorithm searches for an appropriate backtracking point,

Figure 4: DPOR using Clock Vectors.

```
0     Initially: Explore(∅, λx.⊥);

1     Explore(S, C) {
2       let s = last(S);
3       for all processes p {
4         if ∃i = max({i ∈ dom(S) | S_i is dependent
              and may be co-enabled with next(s, p)
              and i ≰ C(p)(proc(S_i))})
          {
5           if (p ∈ enabled(pre(S, i)))
6           then add p to backtrack(pre(S, i));
7           else add enabled(pre(S, i)) to backtrack(pre(S, i));
8         }
9       }
10      if (∃p ∈ enabled(s)) {
11        backtrack(s) := {p};
12        let done = ∅;
13        while (∃ p ∈ (backtrack(s) \ done)) {
14          add p to done;
14.1        let t = next(s, p);
14.2        let S' = S.t;
14.3        let cv = max{C(i) | i ∈ 1..|S| and
                                S_i dependent with t};
14.4        let cv2 = cv[p := |S'|];
14.5        let C' = C[p := cv2, |S'| := cv2];
15          Explore(S', C');
16        }
17      }
18    }
```

Figure 5: DPOR without Stack Traversals.

```
0     Initially: Explore(∅, λx.⊥, λx.0);

1     Explore(S, C, L) {
2       let s = last(S);
3       for all processes p {
4         let i = L(α(next(s, p)));
          if i ≠ 0 and i ≰ C(p)(proc(S_i))
          {
5           if (p ∈ enabled(pre(S, i)))
6           then add p to backtrack(pre(S, i));
7           else add enabled(pre(S, i)) to backtrack(pre(S, i));
8         }
9       }
10      if (∃p ∈ enabled(s)) {
11        backtrack(s) := {p};
12        let done = ∅;
13        while (∃ p ∈ (backtrack(s) \ done)) {
14          add p to done;
14.1        let S' = S.next(s, p);
14.2        let o = α(next(s, p));
14.3        let cv = max(C(p), C(o))[p := |S'|];
14.4        let C' = C[p := cv, o := cv];
14.5        let L' = if next(s, p) is a release
                     then L
                     else L[o := |S'|];
15          Explore(S', C', L');
16        }
17      }
18    }
```

and can be implemented as a sequential search through the transition stack S. The worst-case time complexity of this algorithm is $O(m.d.r)$.

The following invariants hold on each call to Explore: for all $i, j \in dom(S)$ and for all $p \in \mathcal{P}$:

$$\begin{array}{lll} i \rightarrow_S p & \text{iff} & i \leq C(p)(proc(S_i)) \\ i \rightarrow_S j & \text{iff} & i \leq C(j)(proc(S_i)) \end{array}$$

Using these invariants, we can show that the algorithm of Figure 4 is a specialized version of the algorithm of Figure 3 (although it may conservatively add more backtracking points in line 7).

4.2 Avoiding Stack Traversals

This section refines the previous algorithm to avoid traversing the entire transition stack S. Instead, we assume that each transition t operates on exactly one shared object, which we denote by $\alpha(t) \in \mathit{Object}$. In addition, we assume that two transitions t_1 and t_2 are dependent if and only if they access the same communication object, that is, if $\alpha(t_1) = \alpha(t_2)$. Under these assumptions, the dependence relation is an equivalence relation and all accesses to an object o are totally ordered by the happens-before relation.

In this case, it is sufficient to only keep a clock vector $C(o)$ for the *last* access to each object o. The use of per-object instead of per-transition clock vectors significantly reduces the time and space requirements of our algorithm. Maintaining these per-object clock vectors requires $O(m)$ time per explored transition, as shown in lines 14.3–14.4 of Figure 5.

To avoid a stack traversal for identifying backtracking points, we also make some assumptions about the co-enabled relation. Specifically, for any object o that is not a lock, we assume that any two transitions that access o may be co-enabled. (Even if two operations are never co-enabed, it is still safe to assume that they may be co-enabled – this may limit the amount of reduction obtained, but will not affect correctness.) We use an auxiliary variable $L(o)$ to track the index of the last transition that accessed o. When we consider a subsequent access to o by a transition $next(s, p)$, we need to find the last dependent, co-enabled transition that does not happen-before p. By our assumptions, the last access $L(o)$ must be co-enabled and dependent with $next(s, p)$, as they both access the same object o, which is not a lock. Therefore, $L(o)$ is the appropriate backtracking point, provided $L(o)$ does not happen-before p. In the case where $L(o)$ happens-before p, since the accesses to o are totally-ordered, there cannot be any previous access to o that does not happen-before p, and therefore no backtracking point is necessary.

For a lock acquire, the appropriate backtracking point is not the preceding release, since an acquire and a release on the same lock are never co-enabled. Instead, the appropriate backtracking point for a lock acquire is actually the preceding lock acquire. Hence, for any lock o, we use $L(o)$ to record the index of the last acquire (if any) on that lock.

Figure 5 contains a model checking algorithm based on these ideas. On each call to Explore, finding backtracking points requires constant time per process, or $O(m)$ time per explored transition. The clock vector operations on lines 14.3–14.4 also require $O(m)$ time per explored transition. Thus, the overall time complexity of this algorithm is $O(m.r)$. In the following section, we evaluate the performance of this optimized algorithm.

We can show that this algorithm implements Figure 3, based on the following invariants that hold on each call to Explore: for all $i \in dom(S)$, for all $p \in \mathcal{P}$, and for all $o \in Object$:

$$i \rightarrow_S p \text{ iff } i \leq C(p)(proc(S_i))$$
$$L(o) = max\{i \in dom(S) \mid \alpha(S_i) = o \text{ and } S_i \text{ is not a release}\}$$

Note that this implementation supports arbitrary communication objects such as shared variables, but it requires that all operations on shared variables are dependent. In particular, it does not exploit the fact that two concurrent reads on a shared variable can commute. We could improve the algorithm along these lines by recording two clock vectors for each shared variable, one for read accesses and for write accesses, and by using additional data structures to correctly identify backtracking points. The time complexity of the resulting algorithm is $O(m^2.r)$. Due to space constraints we do not present this algorithm.

5. EXPERIMENTAL EVALUATION

In this section, we present a preliminary performance comparison of three different partial-order reduction algorithms:

- *No POR*: A straightforward model-checking algorithm with no partial-order reduction.
- *Static POR*: A high-precision stubborn-set-like algorithm for statically computing persistent sets based on a precise static analysis of the program.
- *Dynamic POR*: Our new dynamic partial-order reduction algorithm for multi-threaded programs shown in Figure 5.

We describe the impact of using sleep sets in conjunction with each algorithm. We also show the benefit of extending *No POR* and *Static POR* to perform a *stateful search*, where visited states are stored in memory and the model checking algorithm backtracks whenever it visits a previously-visited state. We do not have a *Dynamic POR* implementation that supports a stateful search, since it is not obvious how to combine these ideas. Thus, we have ten model checking configurations, and we evaluate each model checking configuration on two benchmark programs.

5.1 Indexer Benchmark

Our first benchmark is the Indexer program of Figure 1, where each threads inserts 4 messages into the shared hash table. For this benchmark, since a static analysis cannot reasonably predict with sufficient accuracy the conditions under which hash table collisions would occur, *Static POR* yields the same performance as *No POR*. For clarity, we do not show the results for *No POR*, since they are identical to those for *Static POR*. Our experimental results are presented in Figure 6. The key for this figure is the same as in Figure 8: we use triangles for *Dynamic POR*, circles for *Static POR*, squares for *No POR*, dotted lines to indicate a stateful search, and hollow objects to indicate the use of sleep sets. Run-time is directly proportional to the number of explored transitions in all these experiments.

For configurations with up to 11 threads, since there are no conflicts in the hash table and each thread accesses different memory locations, the reduced state space with *Dynamic POR* is a single path. In comparison, the *Static POR* quickly suffers from state explosion. When combined with sleep sets, *Static POR* performs better, but still cannot avoid state explosion. Using a stateful search in addition to sleep sets does not significantly further reduce the number of explored transitions.

5.2 File System Benchmark

Our second example in Figure 7 is derived from a synchronization idiom found in the Frangipani file system [24], and illustrates the statically-hard-to-predict use of shared memory that motivates this work. For each file, this example keeps a data structure called an `inode` that contains a pointer to a disk block that holds the file data. Each disk block `b` has a busy bit indicating whether the block has been allocated to an inode. Since the file system is multi-threaded, these data structures are guarded by mutual exclusive locks. In particular, distinct locks `locki[i]` and `lockb[b]` protect each inode `inode[i]` and block busy bit `busy[b]`, respectively. The code for each thread picks an inode `i` and, if that inode does not already have an associated block, the thread searches for a free block to allocate to that inode. This search starts at an arbitrary block index, to avoid excessive lock contention.

Figure 8 shows the number of transitions executed when model checking this benchmark for 1 to 26 threads, using each of the ten model checking algorithms. For *No POR*, the search space quickly explodes, although sleep sets and a stateful search provide some benefit. *Static POR* identifies that all accesses to the `inode` and `busy` arrays are protected by the appropriate locks, thus reducing the number of interleavings explored. Again, sleep sets help, but a stateful search does not provide noticable additional benefit once sleep sets are used. Indeed, the two lines in Figure 8 are essentially identical.

Static POR must conservatively consider that acquires of the locks `locki[i]` may conflict, and similarly for `lockb[b]`. In contrast, *Dynamic POR* dynamically detects that such conflicts do not occur for up to 13 threads, thus reducing the search space to a single path. For larger numbers of threads, since conflicts do occur, sleep sets provide additional benefits in combination with *Dynamic POR*.

5.3 Discussion

The results obtained with the two previous benchmarks clearly demonstrate that our dynamic partial-order reduction approach can sometimes significantly outperform prior partial-order reduction techniques.

However, note that *Dynamic POR* is not always strictly better than *Static POR*, since *Dynamic POR* arbitrarily picks the initial transition t from each state, and then dynamically computes a persistent set that includes t. In contrast, *Static POR* may be able to compute a smaller persistent set that need not include t. Since *Static POR* and *Dynamic POR* are *compatible*, they can be used simultaneously and benefit from each other's strengths – these experiments simply show that *Dynamic POR* can go much beyond *Static POR* in cases where the latter is helpless.

Figure 8: Number of transitions explored for the File System Benchmarks.

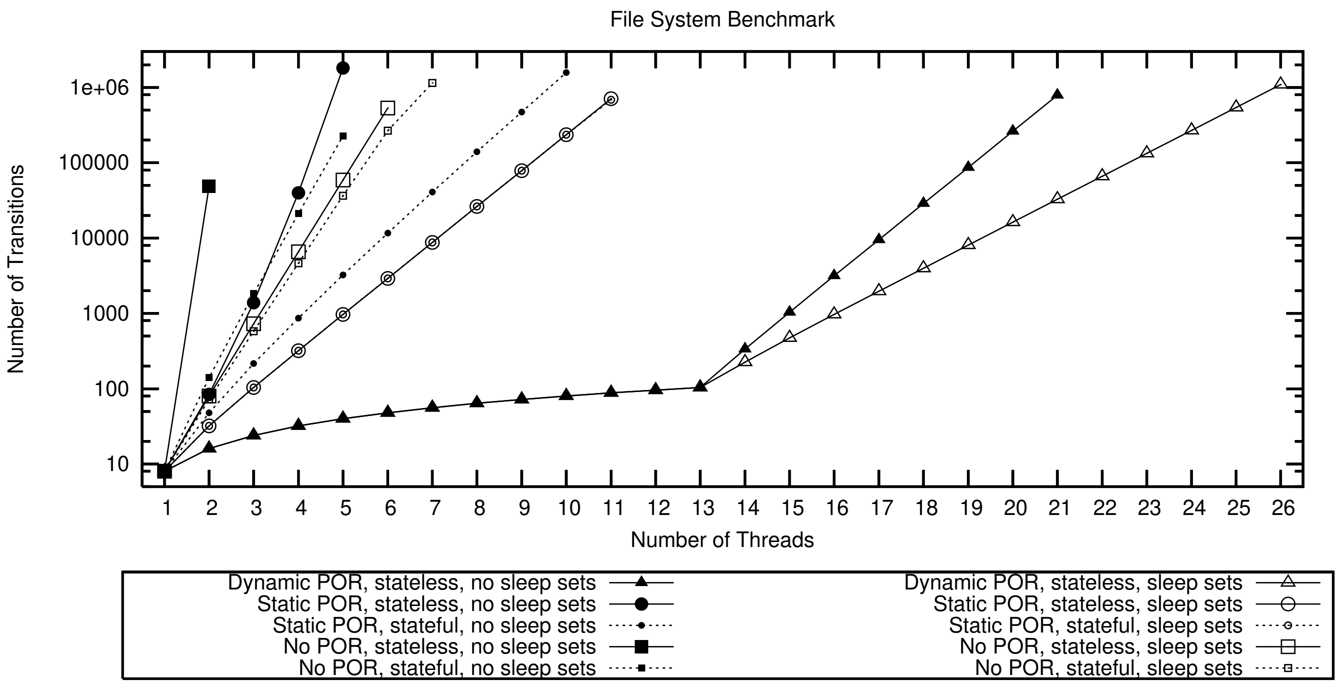


Figure 6: Indexer Benchmark. (See Fig. 8 for key.)

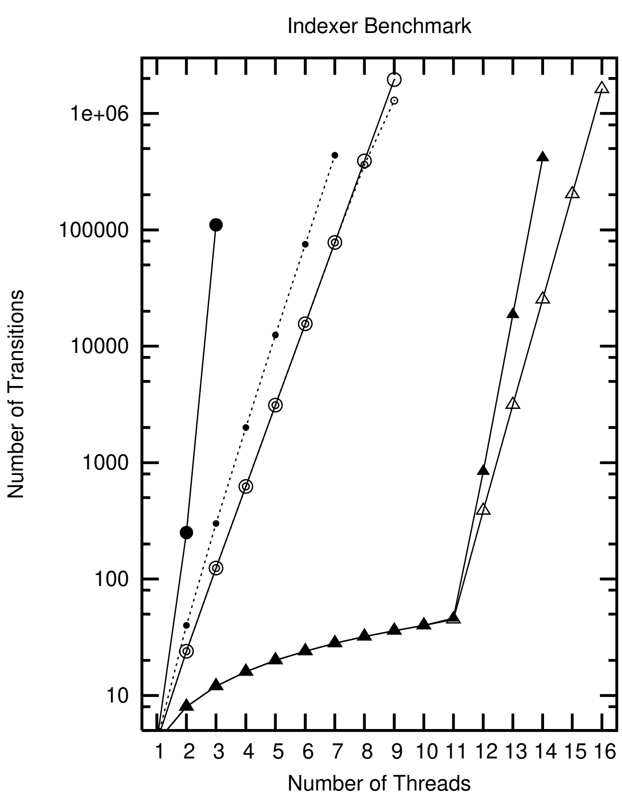


Figure 7: File System Example.

Global variables:

```
const int NUMBLOCKS = 26;
const int NUMINODE = 32;
boolean[NUMINODE] locki;
int[NUMINODE] inode;
boolean[NUMBLOCKS] lockb;
boolean[NUMBLOCKS] busy;
```

Thread-local variables:

```
int i, b;
```

Code for thread `tid`:

```
i := tid % NUMINODE;
acquire(locki[i]);
if (inode[i] == 0) {
    b := (i*2) % NUMBLOCKS;
    while (true) {
        acquire(lockb[b]);
        if (!busy[b]) {
            busy[b] := true;
            inode[i] := b+1;
            release(lockb[b]);
            break;
        }
        release(lockb[b]);
        b := (b+1)%NUMBLOCKS;
    }
}
release(locki[i]);
exit();
```

6. RELATED WORK

Our dynamic partial-order reduction technique has some general similarities with the "least-commitment search strategy" used in non-linear planners (e.g., see [3]) which originally inspired the work on partial-order reduction via net-unfoldings [19], later extended from deadlock detection to full model checking [6, 7]. Loosely speaking, the term "least-commitment strategy" means that every enabled transition is assumed by default not to interfere with any other concurrent transition, unless this assumption is proved wrong later during the search. The net-unfolding technique uses an elaborate data structure, called net-unfolding, for representing explicitly all the partial-order executions explored so far plus all the nondeterminism (branching) to go from one to the other. In contrast, our technique only uses a partial-order representation $\rightarrow_S$ of a single execution trace S. Another key difference is that detecting deadlocks in a net-unfolding is itself NP-hard in the size of the net-unfolding in general [19], while checking whether the current state is a deadlock is immediate with an explicit state-space exploration, as in our approach. We are not aware of any implementation of the net-unfolding technique for languages (like C or Java) more expressive than Petri-net-like formalisms. It would be worth further comparing both approaches.

A number of recent techniques have considered various kinds of *exclusive access predicates* for shared variables that specify synchronization disciplines such as "this variable is only accessed when holding its protecting lock" [21, 22, 8, 5]. These exclusive access predicates can be leveraged to reduce the search space, while simultaneously being verified or inferred during reduced state-space exploration. However, these techniques do not work well when the synchronization discipline changes during program execution, such as when an object is initialized by its allocating thread without synchronization, and subsequently shared in a lock-protected manner by multiple threads. Also, these techniques would not help in the case of the examples considered in the previous section. Note that dynamic partial-order reduction is also compatible and complementary with these techniques.

Partial-order representations of execution traces [16] have also been used for detecting invariant violations in distributed systems (e.g., see [23]). In contrast with this prior work, we exploit partial-order information to determine the possible existence of execution traces that are *not* part of the current partial-order execution, and to introduce backtracking points accordingly in order to prune the state space safely for verification purposes.

7. CONCLUSIONS

We have presented a new approach to partial-order reduction for model checking software. This approach is based on *dynamically* tracking interactions between concurrent processes/threads at run time, and then exploiting this information using a new partial-order reduction algorithm to identify backtracking points where alternative paths in the state space need to be explored.

In comparison to static partial-order methods, our algorithm is easy to implement and does not require a complicated and approximate static analysis of the program. In addition, our dynamic POR approach can easily accommodate programming constructs that dynamically change the structure of the program, such as the dynamic creation of additional processes or threads, dynamic memory allocation, or the dynamic creation of new communication channels between processes. In contrast, static analysis of such constructs is often difficult or overly-approximate.

We therefore believe that the idea of dynamic partial-order reduction is significant since it provides an attractive and complementary alternative to the three known families of partial-order reduction techniques, namely persistent/stubborn sets, sleep sets and net unfoldings.

The algorithms presented in this paper can be used to prune acyclic state spaces while detecting deadlocks and safety-property violations without any risk of incompleteness. In practice, these algorithms can be used for systematically and efficiently testing the correctness of any concurrent software system, whether its state space is acyclic or not. However, in the presence of cycles, the depth of the search has to be bounded somehow, by simply using some arbitrary bound [10] for instance.

For application domains and sizes where computing canonical representations for visited system states is tractable, such representations could be stored in memory and used both to avoid the re-exploration of previously visited states and to detect cycles. It would be worth studying how to combine the type of dynamic partial-order reduction introduced in this paper with techniques for storing states in memory and with existing partial-order reduction techniques for dealing with cycles, liveness properties, and full temporal-logic model checking (e.g., [26, 9, 2]).

Acknowledgements: We thank the anonymous reviewers for their helpful comments. This work was funded in part by NSF CCR-0341658 and NSF CCR-0341179.

8. REFERENCES

[1] T. Ball and S. Rajamani. The SLAM Toolkit. In *Proceedings of CAV'2001 (13th Conference on Computer Aided Verification)*, volume 2102 of *Lecture Notes in Computer Science*, pages 260–264, Paris, July 2001. Springer-Verlag.

[2] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.

[3] P. R. Cohen and E. A. Feigenbaum. *Handbook of Artificial Intelligence*. Pitman, London, 1982.

[4] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, Robby, and H. Zheng. Bandera: Extracting Finite-State Models from Java Source Code. In *Proceedings of the 22nd International Conference on Software Engineering*, 2000.

[5] M. B. Dwyer, J. Hatcliff, V. R. Prasad, and Robby. Exploiting Object Escape and Locking Information in Partial Order Reduction for Concurrent Object-Oriented Programs. *To appear in Formal Methods in System Design*, 2004.

[6] J. Esparza. Model Checking Using Net Unfoldings. *Science of Computer Programming*, 23:151–195, 1994.

[7] J. Esparza and K. Heljanko. Implementing LTL model checking with net unfoldings. In *Proceedings of the 8th SPIN Workshop (SPIN'2001)*, volume 2057 of *Lecture Notes in Computer Science*, pages 37–56, Toronto, May 2001. Springer-Verlag.

[8] C. Flanagan and S. Qadeer. Transactions for Software Model Checking. In *Proceedings of the Workshop on Software Model Checking*, pages 338–349, June 2003.

[9] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems – An Approach to the State-Explosion Problem*, volume 1032 of *Lecture Notes in Computer Science*. Springer-Verlag, January 1996.

[10] P. Godefroid. Model Checking for Programming Languages using VeriSoft. In *Proceedings of POPL'97 (24th ACM Symposium on Principles of Programming Languages)*, pages 174–186, Paris, January 1997.

[11] P. Godefroid. Software Model Checking: The VeriSoft Approach. *To appear in Formal Methods in System Design*, 2005. Also available as Bell Labs Technical Memorandum ITD-03-44189G.

[12] P. Godefroid and D. Pirottin. Refining dependencies improves partial-order verification methods. In *Proceedings of CAV'93 (5th Conference on Computer Aided Verification)*, volume 697 of *Lecture Notes in Computer Science*, pages 438–449, Elounda, June 1993. Springer-Verlag.

[13] P. Godefroid and P. Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. *Formal Methods in System Design*, 2(2):149–164, April 1993.

[14] T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy Abstraction. In *Proceedings of the 29th ACM Symposium on Principles of Programming Languages*, pages 58–70, Portland, January 2002.

[15] S. Katz and D. Peled. Defining conditional independence using collapses. *Theoretical Computer Science*, 101:337–359, 1992.

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

[17] F. Mattern. Virtual Time and Global States of Distributed Systems. In *Proc. Workshop on Parallel and Distributed Algorithms*, pages 215–226. North-Holland / Elsevier, 1989.

[18] A. Mazurkiewicz. Trace theory. In *Petri Nets: Applications and Relationships to Other Models of Concurrency, Advances in Petri Nets 1986, Part II; Proceedings of an Advanced Course*, volume 255 of *Lecture Notes in Computer Science*, pages 279–324. Springer-Verlag, 1986.

[19] K. McMillan. Using unfolding to avoid the state explosion problem in the verification of asynchronous circuits. In *Proc. 4th Workshop on Computer Aided Verification*, volume 663 of *Lecture Notes in Computer Science*, pages 164–177, Montreal, June 1992. Springer-Verlag.

[20] D. Peled. All from one, one for all: on model checking using representatives. In *Proc. 5th Conference on Computer Aided Verification*, volume 697 of *Lecture Notes in Computer Science*, pages 409–423, Elounda, June 1993. Springer-Verlag.

[21] S. D. Stoller. Model-Checking Multi-Threaded Distributed Java Programs. *International Journal on Software Tools for Technology Transfer*, 4(1):71–91, October 2002.

[22] S. D. Stoller and E. Cohen. Optimistic Synchronization-Based State-Space Reduction. In H. Garavel and J. Hatcliff, editors, *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2619 of *Lecture Notes in Computer Science*, pages 489–504. Springer-Verlag, April 2003.

[23] S. D. Stoller, L. Unnikrishnan, and Y. A. Liu. Efficient Detection of Global Properties in Distributed Systems Using Partial-Order Methods. In *Proceedings of the 12th Conference on Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 264–279, Chicago, July 2000. Springer-Verlag.

[24] C.A. Thekkath, T. Mann, and E.K. Lee. Frangipani: A scalable distributed file system. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 224–237, October 1997.

[25] A. Valmari. Stubborn sets for reduced state space generation. In *Advances in Petri Nets 1990*, volume 483 of *Lecture Notes in Computer Science*, pages 491–515. Springer-Verlag, 1991.

[26] A. Valmari. On-the-fly verification with stubborn sets. In *Proc. 5th Conference on Computer Aided Verification*, volume 697 of *Lecture Notes in Computer Science*, pages 397–408, Elounda, June 1993. Springer-Verlag.

[27] W. Visser, K. Havelund, G. Brat, and S. Park. Model Checking Programs. In *Proceedings of ASE'2000 (15th International Conference on Automated Software Engineering)*, Grenoble, September 2000.

APPENDIX: Proof of Theorem 1

Let A_G denote the state space of the system being analyzed, and let s_0 denote its unique initial state.

Define $E(S,i,p)$ as:

$$\begin{array}{l} \{\ q \in \mathit{enabled}(\mathit{pre}(S,i))\ \mid \\ \quad q = p \text{ or} \\ \quad \exists j \in \mathit{dom}(S) : j > i \text{ and } q = \mathit{proc}(S_j) \text{ and } j \rightarrow_S p\} \end{array}$$

Define $PC(S,j,p)$ as:

$$\begin{array}{l} \text{if} \\ \quad S \text{ is a transition sequence from } s_0 \text{ in } A_G \\ \quad \text{and } i = \mathit{max}(\{i \in \mathit{dom}(S) \mid S_i \text{ is dependent and} \\ \quad \text{co-enabled with } \mathit{next}(\mathit{last}(S),p) \text{ and } i \not\rightarrow_S p\}) \\ \quad \text{and } i \leq j \\ \text{then} \\ \quad \text{if } E(S,i,p) \neq \emptyset \\ \quad \text{then } \mathit{backtrack}(\mathit{pre}(S,i)) \cap E(S,i,p) \neq \emptyset \\ \quad \text{else } \mathit{backtrack}(\mathit{pre}(S,i)) = \mathit{enabled}(\mathit{pre}(S,i)) \end{array}$$

Define the postcondition PC for Explore(S) as:

$$\forall p\ \forall w:\ PC(S.w, |S|, p)$$

We first show that the set of transition explored from each reached state is a persistent set, provided the postcondition holds for each recursive call to Explore($\cdot$).

LEMMA 1. *Whenever a state s reached after a transition sequence S is backtracked during the search performed by the algorithm of Figure 3, the set T of transitions that have been explored from s is a persistent set in s, provided the postcondition PC holds for every recursive call Explore($S.t$) for all $t \in T$.*

PROOF. Let

$$\begin{array}{rcl} s & = & \mathit{last}(S) \\ T & = & \{\mathit{next}(s,p) \mid p \in \mathit{backtrack}(s)\} \end{array}$$

We proceed by contradiction, and assume that there exist $t_1, \ldots, t_n \notin T$ such that:

1. $S.t_1 \ldots t_n$ is a transition sequence from s_0 in A_G and
2. $t_1, \ldots, t_{n-1}$ are all independent with T and
3. t_n is dependent with some $t \in T$.

By property of independence, this implies that t is enabled in the state $\mathit{last}(S.t_1 \ldots t_{n-1})$ and hence co-enabled with t_n. Without loss of generality, assume that $t_1 \ldots t_n$ is the *shortest* such sequence. We thus have that

$$\forall 1 \leq i < n : i \rightarrow_{t_1 \ldots t_{n-1}} n$$

(If this was not true for some i, the same transition sequence without i would also satisfy our assumptions and be shorter.) Let w denote the resulting (possibly empty) transition sequence produced by removing from $t_1 \ldots t_{n-1}$ all the transitions t_i (if any) such that

$$i \not\rightarrow_{t_1 \ldots t_{n-1}} \mathit{proc}(t_n)$$

By definition, $S.w$ is itself a transition sequence from s_0 in A_G and we have

$$\begin{array}{rl} & next(last(S.w), proc(t_n)) \\ = & next(last(S.t_1 \ldots t_n), proc(t_n)) \\ = & t_n \end{array}$$

(Although t_n is enabled in $last(S.t_1 \ldots t_{n-1})$, t_n may no longer be enabled in $last(S.w)$, but this does not matter for the proof.)

If $proc(t) = proc(t_n)$ then

$$\begin{array}{rcl} t & = & next(last(S), proc(t)) \\ & = & next(last(S.w), proc(t)) \\ & = & t_n \end{array}$$

since t is independent with all the transitions in w, contradicting that $t_n \notin T$. Hence $proc(t) \neq proc(t_n)$.

Since t is in a different process than t_n and since t is independent with all the transitions in w, we have

$$\begin{array}{rcl} t_n & = & next(last(S.w), proc(t_n)) \\ & = & next(last(S.w.t), proc(t_n)) \\ & = & next(last(S.t.w), proc(t_n)) \end{array}$$

Let $i = |S| + 1$. Consider the postcondition

$$PC(S.t.w, i, proc(t_n))$$

for the recursive call Explore($S.t$). Clearly,

$$i \not\rightarrow_{S.t.w} proc(t_n)$$

(since t is in a different process than t_n and since t is independent with $t_1, \ldots, t_{n-1}$). In addition, we have (by definition of E):

$$\begin{array}{l} E(S.t.w, i, proc(t_n)) \subseteq \\ \quad \{proc(t_1), \ldots, proc(t_{n-1}), proc(t_n)\} \cap enabled(s) \end{array}$$

Moreover, we have by construction:

$$\forall j \in dom(S.w) : j > i \Rightarrow j \rightarrow_{S.t.w} proc(t_n)$$

Hence, by the postcondition PC for the recursive call Explore($S.t$), either $E(S.t.w, i, proc(t_n))$ is nonempty and at least one process in $E(S.t.w, i, proc(t_n))$ is in $backtrack(s)$, or $E(S.t.w, i, proc(t_n))$ is empty and all the processes enabled in s are in $backtrack(s)$. In either cases, at least one transition among $\{t_1, \ldots, t_n\}$ is in T. This contradicts the assumption that $t_1, \ldots, t_n \notin T$.

□

We now turn to the proof of Theorem 1.

Theorem 1. *Whenever a state s reached after a transition sequence S is backtracked during the search performed by the algorithm of Figure 3 in an acyclic state space, the postcondition PC for Explore(S) is satisfied, and the set T of transitions that have been explored from s is a persistent set in s.*

Proof. Let

$$\begin{array}{rcl} s & = & last(S) \\ T & = & \{next(s, p) \mid p \in backtrack(s)\} \end{array}$$

The proof is by induction on the order in which states are backtracked.

(Base case) Since the state space A_G is acyclic and since the search is performed in depth-first order, the first backtracked state must be a deadlock where no transition is enabled. Therefore, the postcondition for that state becomes $\forall p : PC(S, |S|, p)$, and is directly established by lines 3–9 of the algorithm of Figure 3.

(Inductive case) We assume that each recursive call to Explore($S.t$) satisfies its postcondition. That T is a persistent set in s then follows by Lemma 1. We show that Explore(S) ensures its postcondition for any p and w such that $S.w$ is a transition sequence from s_0 in A_G.

1. Suppose some transition in w is dependent with some transition in T. In this case, we split w into $X.t.Y$, where all the transitions in X are independent with all the transitions in T and t is the first transition in w that is dependent with some transition in T. Since T is a persistent set in s, t must be in T (otherwise, T would not be persistent in s). Therefore, t is independent with all the transitions in X. By property of independence, this implies that the transition sequence $t.X.Y$ is executable from s. By applying the inductive hypothesis to the recursive call Explore($S.t$), we know
   $\forall p : PC(S.t.X.Y, |S| + 1, p)$
   which implies (by the definition of PC) that
   $\forall p : PC(S.t.X.Y, |S|, p)$
   Since t is independent with all the transitions in X, we also have that
   $$\forall i \in dom(S.t.X.Y) : i \rightarrow_{S.t.X.Y} p \text{ iff } \quad i \rightarrow_{S.X.t.Y} p$$
   Therefore, by definition,
   $$PC(S.t.X.Y, |S|, p) \text{ iff } PC(S.X.t.Y, |S|, p)$$
   We can thus conclude that
   $\forall p : PC(S.X.t.Y, |S|, p)$
2. Suppose that all the transitions in w are independent with all the transitions in T and $p \in backtrack(s)$. Then
   (a) $next(s, p) \in T$;
   (b) $next(s, p)$ is independent with w;
   (c) p is a different process from any transition in w;
   (d) $next(last(S.w), p) = next(last(S), p)$;
   (e) $\forall i \in dom(S) : i \rightarrow_{S.w} p$ iff $i \rightarrow_S p$.
   Thus, we have $PC(S.w, |S|, p)$ iff $PC(S, |S|, p)$, and the latter is directly established by the lines 3–9 of the algorithm for all p.
3. Suppose that all the transitions in w are independent with all the transitions in T and $p \notin backtrack(s)$. Pick any $t \in T$. We then have that
   (a) $proc(t) \neq p$;
   (b) t independent with all the transitions in w;
   (c) $next(last(S.w), p) = next(last(S.t.w), p)$;
   (d) $\forall i \in dom(S) : i \rightarrow_{S.w} p$ iff $i \rightarrow_{S.t.w} p$.
   Thus, we have $PC(S.w, |S|, p)$ iff $PC(S.t.w, |S|, p)$. By applying the inductive hypothesis to the recursive call Explore($S.t$), we know
   $\forall p : PC(S.t.w, |S| + 1, p)$
   which implies (by the definition of PC) that
   $\forall p : PC(S.t.w, |S|, p)$
   which in turn implies
   $\forall p : PC(S.w, |S|, p)$
   as required.

□