# Partial Order Reduction Without the Proviso

***Ratan Nalumasu***
***Ganesh Gopalakrishnan***

UUCS-96-008

Department of Computer Science
University of Utah
Salt Lake City, UT 84112 USA

August 6, 1996

## *Abstract*

In this paper, we present a new partial order reduction algorithm that can help reduce both space and time requirements of on-the-fly explicit enumeration based verifiers. The partial order reduction algorithms described in [God95, HP94, Pel94, Pel96] were observed to yield very little savings in many practical examples. The reason was traced to the *proviso* in these algorithms that often caused their search to generate many unnecessary states. Our algorithm, called the two-phase algorithm, avoids the proviso, and follows an execution strategy consisting of alternating phases of *partial order reduction of deterministic states* and *depth-first search*. In this paper, we describe the two-phase algorithm, prove its correctness, describe a new verification tool employing it, and provide a number of significant examples, including directory based protocols of a multiprocessor, that demonstrate the superior performance of the two-phase algorithm.

# 1 Introduction

To motivate the problem studied here, consider two processes P and Q where P is `while(1) x++` and Q is `while(1) y++`. If `x` and `y` are local variables, P and Q are executed using the interleaving semantics, and if all the safety properties of interest are of the form $p(x)$ or $q(y)$ (i.e. binary relations such as $x < y$ are not of interest), then it is not necessary to execute the concurrent actions `x++` and `y++` in both orders; i.e., it suffices to either execute `x++` followed by `y++`, or vice versa. This technique (and suitable generalizations thereof) of avoiding some of the possible interleavings of independent actions is known as *partial order reduction* [God95, GP93, HP94, Pel93, Pel94, Pel96, Val90, Val93] whose semantic underpinnings can be traced to Mazurkiewicz traces [Maz89].

In general, a straightforward realization of state-space enumeration based on the interleaving model of concurrency executes all transitions enabled at every state even when the transitions are pairwise independent (soon to be defined formally). Clearly this can result in a state explosion. Partial order reduction methods, such as employed in [God95, HP94], attempt to mitigate this problem by executing only a subset of these transitions (and postponing the rest) *without affecting the truth values* of the properties being verified. However, a naive implementation of partial order reduction may postpone some of these transitions *indefinitely*, which clearly does not preserve all properties. This problem is referred to as *ignoring problem*. Current implementations of partial order reduction [God95, HP94] solve the ignoring problem by using a proviso, first reported in [Pel94, Pel96]. Proviso ensures that the subset of transitions selected a state do not generate a state that is in the stack maintained by the depth first search (DFS) algorithm. If a subset of transitions satisfying this check cannot be found at a state $S$, then all transitions from $S$ are executed by the DFS algorithm. The provisos used in the two implementations, [God95] and [HP94], differ slightly. [God95] and [HGP92] require that at least one of the transitions do not generate a state in the stack, while [HP94] requires a stronger condition that no transition generates a state in the stack. In addition to solving the ignoring problem, the stronger proviso is sufficient to preserve all stutter free LTL formulae (including the liveness properties) properties, while the weaker check preserves only stutter free safety properties [HGP92,HP94,Pel94,Pel96]. We observed that in a large number of practical examples arising in our problem domain (validation of directory based coherence protocols and server-client protocols), the proviso causes the partial order reduction process to be ineffective. As an example, on *invalidate*, a distributed shared memory protocol (described later), the algorithm of [HP94] aborts its search by running out of memory after generating more than 963,000 states. [God95] algorithm also aborts its search after generating a similar number of states. We believe, based on our intuitions, that protocols of this complexity ought to be easy for on-the-fly explicit enumeration tools to handle (an intuition confirmed by our algorithm that finishes comfortably on this protocol). This paper addresses two questions: (i) whether partial order reduction algorithms to preserve safety properties that avoid the proviso can be developed; (ii) if so, do these algorithms perform better than the algorithms using the proviso on realistic protocols? The answer both questions is "yes", as will be described.

In this paper we present a new partial order reduction algorithm called the two-phase algorithm that avoids the proviso. It follows a simple execution strategy consisting of alternating phases of *partial order reduction of deterministic states* and *depth-first search*. A new verification tool employing the two-phase algorithm has been implemented, and applied on a number of protocols. In general, the two-phase verifier exhibits superior performance than the two algorithms that use the proviso mentioned earlier. In particular, the *invalidate* protocol finishes comfortably generating only 193,389 states.

This paper describes the two-phase algorithm, its correctness proof, and performance statistics. The rest of the paper is organized as follows. Section 2 provides background information about model checking and partial order reduction. Section 3 describes the algorithm presented in [HP94] and the two-phase algorithm. In Section 4 we prove that the two-phase algorithm preserves stutter free safety properties (this section may be skipped during first reading). Section 5 summarizes characteristics of the algorithms on a number of examples. Section 6 provides concluding remarks and plans for future work.

## 2 Background

The tools SPIN (based on [HP94, Pel94, Pel96]) and PO-PACKAGE (based on [God95]) as well as our two-phase verifier use Promela [Hol91] as input language. In Promela, a concurrent program consists of a set of sequential processes communicating via a set of global variables and channels. Channels may have a capacity of zero or more. For zero capacity channels, the rendezvous communication discipline is employed. The state of a sequential process consists of a control state ("program counter") and data state (the state of its local variables). In addition, the process can also access global variables and channels. For the sake of simplicity we assume that a channel is a point to point connection between two processes with a non-zero capacity, i.e., we do not consider the rendezvous communication. In the actual implementation of our verifier, these restrictions have been removed. Any process that attempts to send a message on a full channel blocks until a message is removed from the channel. Similarly, any process attempting to receive a message from an empty channel blocks until a message becomes available on that channel.

We now define the following terms with the aid of Figure 1, where `g` is a global variable, `l` is a local variable, `c` is an output channel and `d` is an input channel for `P`, and guarded commands are written as *if . . . fi*. Similar classifications are employed in other partial-order reduction related works.

**local:** A transition is said to be local if it does not involve a global variable. Examples: `c!7,` `d?l,` `l=0,` `l==0,` `l!=0,` and `skip`.

**global:** A transition is said to be global if it involves a global variable. Example: `g=l`.

**unconditionally safe:** A local transition is said to be unconditionally safe if its executability cannot be changed by the execution of any other process. Examples: `l=0`, `l==0`, `l!=0`.

**conditionally safe:** A local transition is said to be conditionally safe if its executability can be changed by the execution of (a transition of) some other process. Example: `c!7`; when `c` is full, the

```
process P
{
    int l;
    if
        c ! 7 -> skip;
      | d ? l -> skip;
    fi;
    l = 0;
    g = l;
    if
       l == 0 -> skip
     | l != 0 -> skip
    fi;
 }
```

Figure 1: A sample process to illustrate definitions

statement cannot be executed. However, as soon as another process consumes a message from `c`, the statement becomes executable. Hence this is a conditionally safe transition with the condition being that `c` is not full. Another example: `d?l`, with the condition that `d` is not empty.

**safe:** A transition $t$ is safe in a state $S$ if $t$ is an unconditionally safe transition or $t$ is a conditionally safe transition whose condition evaluates to true in $S$.

**internal:** A control state of a process is said to be internal if all the transitions possible from it are local transitions. Example: In Figure 1, the control state corresponding to the first `if` statement is internal since the two transitions possible here, namely `c!7` and `d?l` are local transitions.

**deterministic:** A process $P$ is said to be deterministic in a product state $S$ if $P$ is internal in $S$, written *deterministic(P, S)*, if all transitions from the control state are safe, and exactly one transition of $P$ is executable. Example: In Figure 1, if control state of $P$ is at the second `if` statement, $P$ is deterministic since only one of the two conditions `l==0` and `l!=0` can be true. In general, determining whether a given process is deterministic in a given state requires knowledge of the values of variables and/or contents channels.

**non-deterministic:** A process $P$ is said to be non-deterministic in a state $S$ if $P$ is not deterministic in $S$.

The above definitions are made in order to effect partial-order reduction. For example assume that in a *product* state $S$, $l$ is a safe and executable transition of process $P$ (specifically let $l$ be a receive action on channel $c$ and $c$ has $k > 0$ messages), $m$ is an executable transition of a different process $Q$, and execution of $l$ in $S$ results in a state $S1$. Then, $m$ will be executable in $S1$ for the following reason. Since $m$ was executable in $S$ the only way $m$ can become unexecutable in $S1$ is if $m$ also attempts to receive a message from $c$ and $k$ is 1. But since all channels are assumed to be point-to-point and $m$ is a transition of a different process, $m$ cannot be a receive transition from $c$. Thus $m$

continues to be executable. Therefore, in $S$, it is permissible to consider the interleaving $l; m$ and never consider $m; l$, i.e., $m$ can be postponed in $S$. A similar argument can show that if $l$ is a safe and unexecutable transition, and if execution of $m$ in $S$ results in a state $S2$ then $l$ will be unexecutable in $S2$. Partial order reduction make use of these two properties to reduce the amount of interleaving in the following fashion. Whenever a state $S$ is explored by the partial order reduction algorithm, instead of considering all successors of the $S$, the algorithm attempts to find a process $P$ such that $P$ is in an internal state and all transitions of $P$ from that internal state are safe, and considers the transitions of $P$ only. The algorithm also needs to address the ignoring problem, i.e., care must be taken to ensure that $m$ is not postponed indefinitely.

The two-phase algorithm performs the search in the following way. Whenever a state $S$ is explored by the algorithm, in the first phase all deterministic processes are executed one after the other, resulting in a state $S'$. In the second phase, the algorithm explores *all* transitions enabled at $S'$. The second phase of executing all transitions of $S'$ assures that the ignoring problem is addressed.

# 3 Algorithms

This section provides an overview of the algorithm presented in [HP94] and the two-phase algorithm. The algorithm presented in [HP94] attempts to find a process in an internal state such that all transitions of that process from that internal state are safe and that none of the transitions of the process result in a state that is in the stack. This is the stronger proviso, as pointed out earlier. If a process satisfying the above criterion can be found, then the algorithm examines all the enabled transitions of that process. If no such process can be found, all enabled transitions in that state are examined. In general, an algorithm using the strong proviso generates more states than another algorithm using the weak proviso, since the weak proviso can be satisfied more often than the strong proviso, and any time a process satisfying the above criterion cannot be found, all process in that state are examined by the algorithm. Since the two-phase algorithm is intended to preserve only safety, to obtain an equitable comparison of its performance against that of [HP94] algorithm we implemented the [HP94] algorithm such that the algorithm uses the weaker proviso, and refer to this implementation as "the Proviso algorithm". The proviso algorithm is shown in as *dfs1()* in Figure 2. In this algorithm, *Choose()* is used to find a process satisfying the above criterion. As mentioned earlier, the use of proviso (weak or strong) can cause the algorithm to generates many unnecessary of states. In some protocols, e.g., Figure 3 (a), all reachable states in the protocol are generated. Figure 3 (c) shows the state space generated on this protocol. Another algorithm that uses the (weak) proviso and sleepsets [GHP92], [God95] (implemented in the tool PO-PACKAGE), also exhibits similar state explosion.

The two-phase algorithm is shown as *dfs2()* in Figure 2. In the first phase, *dfs2()* executes deterministic processes. States generated in this phase are saved in the temporary variable `list`. These states are added to `cache` during the second phase. In the second phase, *all* transitions enabled at `s` are examined.

```
initialize stack to contain initial state
initialize cache to contain initial state

dfs1()
{
    s := top(stack);
    (i, found) := Choose (s);
    if (found) {
        tr := {t | t is enabled in s
                    and PID(t)=i};
        nxt := successors of s obtained by
               executing transitions in tr;
    } else {
        tr := all enabled transitions from s;
        nxt := successors of s obtained
               by executing transitions in tr;
    }

    for each succ in nxt do {
        if succ not in cache then
            cache := cache + {succ};
            push(succ, stack);
            dfs1();
    }
    pop(stack);
}
```

```
initialize stack to contain initial state
initialize cache to Φ

dfs2()
{
    s := top(stack);
    list := {s};
    /* Phase I: partial order step */
    for i := 1 to nprocesses {
       while (deterministic(s,i)) {
           /* Execute the only enabled
              transition of process i */
            s := next(i, s);
            if (s ∈ list) goto NEXT_PROC;
            list :=  list + {s};
        }
      NEXT_PROC: /* next i */
    }
   /* Phase II: classical DFS */
    if (s ∉ cache) {
      cache := cache + list;
        nxt := all successors of s;
        for each succ in nxt {
           if (succ ∉ cache)
                push(succ, stack);
                 dfs2();
          }
     } else {
          cache := cache + list;
     }
     pop (stack);
  }
```

Figure 2: dfs1() is a partial order reduction algorithm using proviso. dfs2() is performs avoids proviso using a different execution strategy.

The two-phase algorithm outperforms the proviso algorithm and [God95] algorithm when the proviso is invoked many times; confirmed by the examples in Section 5. In most reactive systems, a transaction typically involves a subset of processes. For example, in a server-client model of computation, a server and a client may communicate without any interruption from other servers or clients to complete a transaction. After the transaction is completed, the state of the system is reset to the initial state. If the partial order reduction algorithm uses the proviso, state resetting cannot be done as the initial state will be in the stack until the entire reachability analysis is completed. Since at least one process is not reset, the algorithm generates unnecessary states, thus increasing the number of states visited. As shown in Figure 3, in certain examples, *dfs1()* generates all the reachable configurations of the systems. In realistic systems also the number of extra states generated due to the proviso can be high. The two-phase algorithm does not use the proviso. Instead it alternates one step of partial order reduction step with one step of complete depth first search. Thus on protocols that have less non-determinism (and hence that have a large number of states that are deterministic with respect to at least one process) and that reach the initial configuration after completion of a transaction perform better with the two-phase algorithm. We have found this to be the case with virtually all the protocols arising in the context of distributed shared memory multiprocessor implementation [Ava]. If, on the other hand, the protocol under consideration has lot of non-determinism, the two-phase algorithm would not perform well.

Appendix B shows a slightly different version of the two-phase algorithm obtained from the following observation. Execution of a process, say $P2$, in the first phase may make $P1$ deterministic. Since the processes are examined in strict order in the `for` loop of the first phase, $P1$ will be not be executed in the first phase. Intuitively, it seems that executing $P1$ in the second phase instead of the first phase generates more states. Hence, it is beneficial to execute $P1$ in the first phase itself. But such a modification to the algorithm results in a more complex control structure. A similar effect can be achieved by modifying the second phase. If the second phase examines only non-deterministic processes, a similar effect can be achieved. An algorithm with this modification to the second phase is shown in Appendix B.

## 4 Correctness of the Two-phase Algorithm

We show that *dfs2()* preserves all stutter free safety properties. *dfs3()* (Figure 7 in Appendix B) can be shown to preserve safety properties along the same lines. To establish the correctness of *dfs2()*, we need the following two lemmas.

**Lemma 1:** A state $S$ is added to `cache` only after ensuring that all transitions enabled at $S$ will be executed at $S$ or at a successor of $S$. This lemma asserts that *dfs2()* does not suffer from ignoring problem.

**Proof:** Proof is based on induction on the "time" a state is entered into `cache`.

*Induction Basis:* During the first call of *dfs2()* the outer "if" statement of the second phase will be executed; during this phase, all states in `list` are added to `cache` in the body of the "if" statement. Following that the algorithm examines all successors of `s`. Let $s'$ be an arbitrary element of `list`. By the manner in which `list` is generated, $s'$ can reach `s` via zero or more deterministic transitions. By the definition of deterministic transition, any transition enabled at $s'$, but not executed in any of the states along the path from $s'$ to `s` will remain executable at `s`. Since all transitions out of `s` are considered in the second phase, it follows that all unexecuted transitions out of $s'$ are also considered. Hence the addition of $s'$ to `cache` satisfies Lemma 1.

*Induction Hypothesis:* Let the states entered into `cache` during the first $i-1$ calls to *dfs2()* be $s_1$, $s_2, \ldots, s_{n-1}$. Assume by induction hypothesis that all transitions enabled at every state $s_i$ in this list are guaranteed to be executed at $s_i$ or a successor of $s_i$.

*Induction Step:* We wish to establish that the states entered into `cache` during the $i^{th}$ call to *dfs2()* also satisfy the Lemma. There are two cases to consider:

1. the outer "if" statement of the second phase is executed
2. the "else" statement of the second phase is executed

In the first case, all successors of `s` are considered in the body of the "if" statement. Let $s'$ be an arbitrary element of `list`. Any transition enabled at a state $s'$ and not taken in the path from $s'$ to `s` is also enabled at `s`. Therefore $s'$ can be added to `cache` without violating the lemma. In the second case, `s` is already in `cache`; it was entered during an earlier call to *dfs2()*. By induction hypothesis, all transitions enabled at `s` are already executed or guaranteed to be executed. Hence all transitions enabled at $s'$ were either already considered at `s` or guaranteed to be executed by the hypothesis. Hence adding $s'$ to `cache` does not violate the lemma. Thus in both cases, `list` can be added to `cache` without violating the lemma.

**Lemma 2:** *dfs2()* terminates after a finite number of calls.

**Proof:** There are two parts to the proof: (a) eventually no new calls to *dfs2()* are made, and (b) the `while` loop in the first phase terminates. To prove (a), note that new calls to *dfs2()* are made only in the body of the outer "if" statement in the second phase. Before these calls are made, all elements of `list` are added to `cache`. The precondition to execute the "if" statement is that `s` is not in `cache`. By construction of `list`, `s` is in `list`. Thus the number of states in `cache` increases at least by one as a result of adding `list` to `cache`. In other words, if number of states in `cache` before the $i^{th}$ level call of *dfs2()* is made is $k$, then the number of states in `cache` before $i+1^{th}$ level call of dfs2() is made is at least $k+1$. Thus the maximum depth of calls to *dfs2()* cannot exceed the number of states in the protocol, which is finite. To prove (b), note that one new state is added to `list` in each iteration of `while` loop. Again, since the number of states in the protocol is finite, eventually no new states can be added to `list`, thus the while loop terminates.

***Theorem 1:*** *dfs2()* finds all safety violations present in the protocol.

***Proof:****(Informal)* The proof of the theorem follows from the observation that finding a safety violation requires that every enabled transition be executed. Further, a transition need not be executed at a state if it is executed at a successor of that state obtained by executing a sequence of safe transitions. *dfs2()* satisfies these two conditions. In particular, *dfs2()* might not execute a transition $t$ from a state $s$ if a safe transition $t^{'}$ is taken from $s$. This can happen during the first phase of *dfs2()* where only deterministic processes are considered (a deterministic process has exactly one enabled transition which is also a safe transition). Lemma 1 guarantees that all enabled transitions at every state are considered by *dfs2()*. Hence, *dfs2()* is safety preserving.

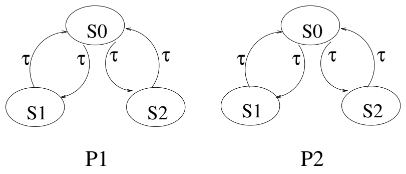
The fact that *dfs3()* preserves safety properties follows from the observation that the transitions in the second phase that would be considered by *dfs2()* but not by *dfs3()* would be considered by *dfs3()* in its first-phase during the subsequent recursive calls of *dfs3()*. [Nal] presents a complete proof that *dfs3()* preserves safety properties.

# 5 Case studies

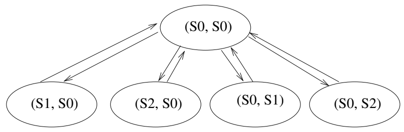
In this section, we present the results of running the proviso algorithm, the two-phase algorithms, and the [God95] algorithm on two artificial protocols and several realistic protocols.

## 5.1 Best case

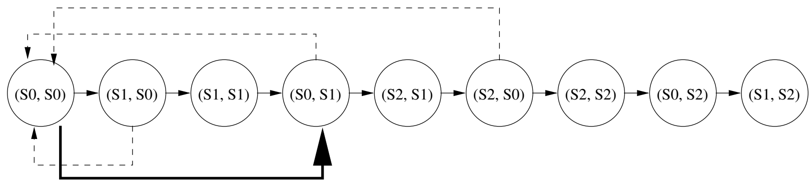
Figure 3 presents an example of the protocol that runs more efficiently with the two-phase algorithms. Table 1 shows the results of running the algorithms on this protocol. On a system comprising of $n$ processes, the two-phase algorithms generates $2n + 1$ states while the proviso algorithm generates $3^n$ states.




(a) Best case




(b) State space by 2 phase




(c) State space generated by proviso algorithm

Figure 3: Best case protocol. (a) The protocol. (b) State space that generated by the 2-phase algorithms. (c) State space generated by SPIN using weak proviso. Dotted lines in (c) show some of the transitions that were not attempted due to proviso. The thick line in (c) shows one of the transitions that would be taken by the algorithm, but find that the state is already generated.

| N | Proviso Algorithm | [God95] Algorithm | First two-phase Algorithm | Second two-phase Algorithm |
|---|---|---|---|---|
| 4 | 81/0.32 | 70/0.35 | 9/0.33 | 9/0.33 |
| 5 | 243/0.34 | 217/0.42 | 11/0.33 | 11/0.33 |
| 6 | 729/0.38 | 683/0.64 | 13/0.33 | 13/0.33 |
| 7 | 2187/0.50 | 2113/1.4 | 15/0.33 | 15/0.33 |
| 8 | 6561/0.83 | 6422/4.34 | 17/0.33 | 17/0.33 |

Table 1: Number of states saved in the hash table, and time taken by different algorithms on Best Case.

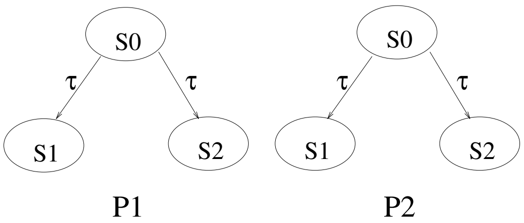



Figure 4: Worst case protocol. Statistics for this protocol are in Table 2.

## 5.2 Worst case

Figure 4 shows an example of the protocol that runs better with the proviso algorithm than the two-phase algorithms. This protocol has a total of $3^n$ states where $n$ is the number of processes in the system. As can be seen, the proviso algorithm can reduce the number of states to $2^{n+1} - 1$ states, while the two-phase algorithms fail to bring any reductions. The reason for the bad performance of two-phase algorithms is that none of the reachable states is deterministic with respect to any process. Hence, the two-phase algorithms degenerate to classical depth first search.

| N | Proviso Algorithm | [God95] Algorithm | First two-phase Algorithm | Second two-phase Algorithm |
|---|---|---|---|---|
| 5 | 63/0.33 | 64/0.37 | 243/0.39 | 243/0.35 |
| 6 | 127/0.39 | 128/0.42 | 729/0.49 | 729/0.45 |
| 7 | 255/0.43 | 256/0.51 | 2187/0.76 | 2187/0.76 |
| 8 | 511/0.43 | 512/0.7 | 6561/1.71 | 6561/1.62 |
| 9 | 1023/0.51 | 1024/1.21 | 19683/4.88 | 19683/4.92 |

Table 2: Number of states saved in the hash table, and time taken by different algorithms on Worst Case.

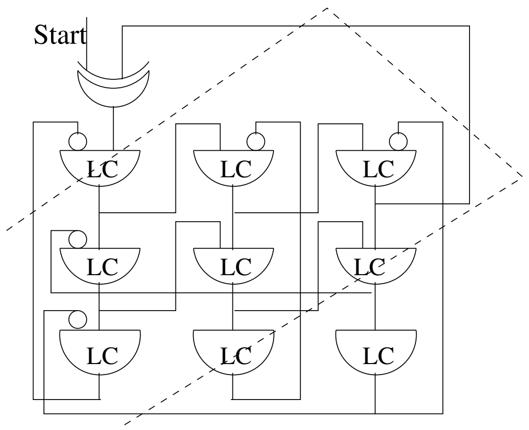



Figure 5: Wavefront arbiter of size 3x3. The dotted line shows one of the three wrapped diagonals. All the lockable C-elements on a wrapped diagonals may operate concurrently to implement the arbitration.

| N | Proviso Algorithm | First two-phase Algorithm | Second two-phase Algorithm |
|---|---|---|---|
| 6 | 281/0.53 | 172/0.42 | 156/0.40 |
| 7 | 384/0.68 | 230/0.47 | 210/0.42 |
| 8 | 503/0.91 | 296/0.56 | 272/0.47 |
| 9 | 638/1.30 | 370/0.69 | 342/0.55 |
| 10 | 789/1.76 | 452/0.89 | 420/0.66 |

Table 3: Number of states saved in the hash table, number of transition traversed and time taken for the reachability analysis of the wavefront arbiter by different algorithms. [God95] algorithm is not reported since it couldn't handle the large number of processes involved.

---

### 5.3 Wavefront Arbiter

A cross-bar arbiter that operates by sweeping diagonally propagating "wavefronts" within a circuit array [Gop94] is shown in Figure 5. To request a cross-bar connection at a location $ij$ a request is placed at the "lockable" C-element [Gop94] at this location. This request attempts to "pin down" the wavefront at this location. When this attempt succeeds, the crossbar connection $ij$ can be used. A property maintained by this arbiter is that no two C-elements on any row or a column can support a wavefront concurrently. This allows the arbiter to support concurrent arbitration requests (e.g., those falling on the wrapped diagonal in the figure) that don't conflict on a row or a column. The results for this protocol are presented in table 3. Statistics for [God95] algorithm is not reported on this example as the protocol contains a large number of processes that the implementation could not handle.

| Protocol | Proviso Algorithm | [God95] Algorithm | First two-phase Algorithm | Second two-phase Algorithm |
|---|---|---|---|---|
| Migratory | 34906/5.08 | 28826/14.45 | 23163/2.84 | 23163/2.84 |
| Invalidate | Unfinished | Unfinished | 193389/19.23 | 193389/19.23 |

Table 4: Number of states saved in the hash table, time taken in seconds for reachability analysis on migratory and invalidate protocols by different algorithms.

---

### 5.4 DSM Protocols

Several realistic directory-based distributed shared memory protocols from the Avalanche multiprocessor project [Ava] underway at the University of Utah were experimented with. Directory based protocols to implement shared memory in multiprocessors are gaining popularity due to the scalable nature of the protocols. In a directory based system, every cache line has a designated *home node*—a processor responsible for maintaining the coherency of that line. Whenever a node tries to access a cache line for reading or writing if the line is not present in the local cache in an appropriate state, a message is sent to the home node of that line. The home node, upon receiving the request may need to contact some or all of the nodes that currently hold the line in their caches. The home node then will supply the data with the required access permissions to the requester.

Some of the well known directory based coherency protocols are *write invalidate*, *write update*, and *migratory*. A brief explanation of these protocols is provided for the sake of completeness. Whenever a cache line managed by the write invalidate protocol is modified by a node, the node sends a message to the home node. The home node in turn invalidates all the nodes holding a copy of the shared line in their caches. In the write update protocol, on the other hand, the new value of the data is broadcast to all the nodes holding a copy of the cache line in their caches. The migratory protocol does not send such updates or invalidate messages, but instead ensures that the line is present in at most one node's cache[1]. Table 4 presents the results of running the different algorithms on the migratory and the invalidate protocols. Migratory protocol contains about 200 lines of Promela code and invalidate protocol contains about 330 lines of Promela code excluding comments. All the verification runs were limited to 64 MB of memory. It can be seen that proviso algorithm did not complete the search on invalidate protocol. This algorithm aborted search after generating more than 963,000 states due to unavailability of more memory. In contrast, the two-phase algorithm completed the search generating only a modest 193,389 states.

[1] This is a simplistic view of the protocol, as the protocol allows a line to be present at multiple nodes for a *short* period of time for the sake of efficiency.

| N | Proviso Algorithm | [God95] Algorithm | First two-phase Algorithm | Second two-phase Algorithm |
|---|---|---|---|---|
| 2 | 295/0.47 | 242/0.47 | 272/0.34 | 169/0.35 |
| 3 | 11186/3.43 | 8639/7.74 | 3232/0.83 | 3037/0.92 |
| 4 | Unfinished | Unfinished | 62025/14.9 | 59421/14.5 |

Table 5: Number of states saved in the hash table, and time taken for reachability analysis on Server/Client protocol by different algorithms.

### 5.5 A Server/Client protocol

A protocol consisting of N servers and N clients was studied. In this protocol, whenever a client is free, it chooses one of the N servers, and starts communicating with the server. A server waits until a message is received from any one of the N clients, and then services the client. A service consists of doing a simple local calculation, sending the result of the computation to the client, waiting for a *terminate* message from the client, and then acknowledging the *terminate* message with another message. The results of running this protocol are presented in Table 5. Proviso algorithm and [God95] algorithm did not finish the search in a total of 64 Megabytes of memory when the protocol consists of 4 servers and 4 clients.

### 5.6 Other Protocols

We also ran the two-phase protocols on the protocols provided as part of SPIN distribution. Some of the protocols supplied with SPIN distribution are not perpetual processes (i.e., they terminate or deadlock). *Sort* protocol in the SPIN distribution terminates after a finite number of steps, and the *snoopy* protocol has a large number of sequences where the protocol deadlocks. *Sort* is a protocol to sort a sequence of numbers. Since this protocol has no non-determinism and terminates after a finite number of steps, the proviso algorithm and two-phase generate equal number of states. *Snoopy* is a cache coherency protocol to maintain consistency in a bus based multiprocessor system. This protocol contains a large number of deadlocks, and therefore the two-phase algorithm is not as effective. *Pftp* is a flow control protocol. This protocol contains little determinacy. Hence two-phase algorithm is not as effective. Run times of these protocols are summarized in Table 6.

## 6 Conclusions

We have presented two closely related new algorithms for partial order reduction to preserve safety properties. Unlike the proviso algorithm or [God95] algorithm, these algorithms do not use proviso.

| Protocol | Proviso Algorithm | [God95] Algorithm | First two-phase Algorithm | Second two-phase Algorithm |
|---|---|---|---|---|
| Sort | 174/0.35 | 173/0.6 | 174/0.33 | 174/0.39 |
| Snoopy | 20323/6.22 | 10311/10.53 | 20186/5.08 | 18842/5.0 |
| Pftp | 161751/34.5 | 125877/150.7 | 230862/36.3 | 230862/39.4 |

Table 6: Number of states saved in the hash table, and time taken for reachability analysis on protocols supplied as part of SPIN distribution by different algorithms.

---

Instead they alternate one step of partial order reduction step (using deterministic moves) with one step of classical depth first search (using all moves). These algorithms are shown to perform better than other the previous algorithms on protocols where the proviso is invoked many times. As shown using case studies, the number of states explored by these algorithms can be substantially less than the number of states explored by other algorithms on reactive systems where the initial state is reached after a transaction is completed. However, in certain cases, the two-phase algorithms may generate more states than the algorithms using proviso. This can happen if the amount of non-determinism is high in the protocol and the proviso is invoked very few times as the protocol terminates or deadlocks.

It is possible to modify the first phase of the two-phase algorithms to make use of all safe transitions instead of using just deterministic transitions. The advantage of such an algorithm would be that, unlike the two-phase algorithms that degenerate to full state search on such protocols as worst-case, this algorithm would degenerate to the proviso algorithm (we have implemented this algorithm whose control structure turns out to be quite complex.) Also, [HP94] preserves all stutter free LTL formulae [Pel94, Pel96] while the two-phase algorithms preserve only stutter free safety properties. At present, it is not clear, if there is a simple variation of the two-phase algorithm that preserves all stutter free LTL formulae.

# A Promela Constructs

Figure 6 illustrates the core constructs of Promela and condition for a statement to be local, executable and if the statement is local, the condition under which the statement is safe.

| Statement | Meaning | local | executable | safe |
|---|---|---|---|---|
| x = E | Assignment | x is local, and E contains only locals | *true* | *true* |
| await(B) | Wait until B becomes true | B contains no global variables | B is *true* | *true* |
| c ! E | Send the value of expression on channel c | E contains no global variables | c is not full | c is not full |
| c ? v | Receive the first message of the channel into variable v | v is local | c is not empty | c is not empty |
| c ? C | Remove the first message from the channel which is constant C | *true* | First message on c is constant C | c is not empty |
| goto label | Assignment to control state | *true* | *true* | *true* |

Figure 6: Core constructs of Promela and conditions under which they are local, executable and safe

## B A more efficient two-phase algorithm

```
initialize stack to contain initial state
initialize cache to Φ

dfs3()
{
    s := top(stack);
    list := {s};
    /* Phase I: partial order step */
    for i := 1 to nprocesses {
        while (deterministic(s,i)) {
            s := next(i, s);
            if (s ∈ list) goto NEXT_PROC;
            list :=  list + {s};
        }
NEXT_PROC: /* next i */
    }
    /* Phase II: classical DFS */
    if (s ∉ cache) {
        cache := cache + list;
        pr := {all processes in non-deterministic state in s};
        tr := {t | all transitions enabled in s such that PID(t) ∈ pr};
        nxt := successors of s obtained by executing transitions in tr;
        call_dfs := true;
        for each succ in nxt {
            if (succ ∉ cache)
                call_dfs := false;
                push(succ, stack);
                dfs3();
        }
        if (call_dfs) {
          push (s, stack);
          dfs3();
        }
    }
    else {
        cache := cache + list;
    }
    pop (stack);
}
```

Figure 7: Two-phase algorithm to implement partial order reduction that avoids the proviso. In the second phase only non-deterministic processes are considered.

## References


[Ava] See http://www.cs.utah.edu/projects/avalanche for details.

[GHP92] Patrice Godefroid, Gerard Holzmann, and Didier Pirottin. State-space caching revisited. In *Computer Aided Verification*, pages 178–191, Montreal, Canada, June 1992.

[God95] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An approach to the State-Explosion Problem*. PhD thesis, Univerite De Liege, 1994–95.

[Gop94] Ganesh Gopalakrishnan. Developing micropipeline wavefront arbiters using lockable C-elements. *IEEE Design & Test of Computers*, 11(4):55–64, Winter 1994.

[GP93] Patrice Godefroid and Didier Pirottin. Refining dependencies improves partial-order verification methods. In *Computer Aided Verification*, pages 438–450, Elounda, Greece, June 1993.

[HGP92] Gerard Holzmann, Patrice Godefroid, and Didier Pirottin. Coverage preserving reduction strategies for reachability analysis. In *International Symposium on Protocol Specification, Testing, and Verification*, Lake Buena Vista, Florida, USA, June 1992.

[Hol91] Gerard Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1991.

[HP94] Gerard Holzmann and Doron Peled. An improvement in formal verification. In *FORTE*, Bern, Switzerland, October 1994.

[Maz89] A. Mazurkiewicz. Basic notions of trace theory. In *Linear Time, Branching Time, and Partial Order in Logics and Models for Concurrency*, volume 354. Springer Verlag, Lecture Notes in Computer Science, 1989.

[Nal] See http://www.cs.utah.edu/˜ratan/verif/two2.html.

[Pel93] Doron Peled. All from one, one for all: On model checking using representatives. In *Computer Aided Verification*, pages 409–423, Elounda, Greece, June 1993.

[Pel94] Doron Peled. Combining partial order reductions with on-the-fly model-checking. In *Computer Aided Verification*, pages 377–390, Stanford, California, USA, June 1994.

[Pel96] Doron Peled. Combining partial order reductions with on-the-fly model-checking. *Journal of Formal Methods in Systems Design*, 8 (1):39–64, 1996.

[Val90] Antti Valmari. A stubborn attack on state explosion. In *Computer Aided Verification*, pages 156–165, New Brunswick, NJ, USA, June 1990.

[Val93] Antti Valmari. On-the-fly verification with stubborn sets. In *Computer Aided Verification*, pages 397–408, Elounda, Greece, June 1993.