A New Partial Order Reduction Algorithm for Concurrent System Verification

Ratan Nalumasu       Ganesh Gopalakrishnan
Department of Computer Science
University of Utah, Salt Lake City, UT 84112
{ratan, ganesh}@cs.utah.edu

Abstract

There is great need for automatic verification tools capable of verifying large concurrent system models that arise in system-level hardware designs. This paper presents a new partial order reduction algorithm called Two-phase that is implemented in the verification tool, PV (Protocol Verifier) that attempts to achieve this goal. Two-phase significantly reduces space and time requirements on many practically important protocols on which the partial order reduction algorithms implemented in previous tools [God95, HP94, Pel93, Pel96] yield very little savings. The lack of performance by all these tools is attributable to their use of a proviso—run-time check—in deciding which processes to run in a given state. Two-phase avoids this proviso and follows a much simpler execution strategy that is easier to prove correct; and also dramatically cuts down the number of executions examined on a significant number of examples.

In this paper, we describe the Two-phase algorithm and prove its correctness. The Two-phase algorithm is implemented in a new verification tool called the Protocol Verifier (PV). We provide a number of realistic examples, including directory based protocols of a multiprocessor, which demonstrate the performance of the Two-phase algorithm.

1 Introduction

With the increasing scale of hardware systems and the corresponding increase in the number of concurrent protocols involved in their design, formal verification of concurrent protocols is an important practical need. Explicit state enumeration methods [CES86, Hol91] have shown considerable promise in verification of real-world protocol verification problems [Dill96, Hol91], and have been used with success on many industrial designs [YGM+95, DPN93]. Using most explicit state enumeration tools, a concurrent system is modeled as a set of concurrent processes communicating via shared variables [Dill96] and/or communication channels [Hol91] executing under an interleaving model. The tool, in effect, creates the relevant execution space and checks for linear-time temporal logic safety and liveness properties [Hol91]. In order to handle state-space sizes that arise in practice, these tools incorporate several optimizations, which include generating only a canonized version of all reachable states (otherwise known as symmetry exploitation [ID93, NG95]), hash-compression of state vectors [Hol91, SD96, WL93], and partial-order reductions [Pel93, Pel96, HGP92, God90, GP93, Val90, Val93].

Partial-order reduction is a natural optimization method for the interleaving model of concurrent execution. It exploits the independence between actions in different processes and thereby avoids exhibiting all possible interleavings between them. As a simple example, consider two concurrent
processes P and Q where P consists of the single command \( x^+ \) and Q consists of \( y^+ \). If \( x \) and \( y \) are local variables, if P and Q are executed using an 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. Partial-order reduction algorithms play a very important role in mitigating state explosion, often reducing the computational cost by an exponential factor. This paper presents a new partial-order reduction algorithm called Two-phase that, in most cases, outperforms all comparable algorithms, and is part of a new protocol verification tool called PV that finds routine application in an ongoing multiprocessor design project [Av].

As proved in this paper, PV preserves all stutter-free safety properties [Lam94]. In other words, the truth value of formulae of the form \( \square P \) where \( P \) is a propositional logic formula that does not refer to the stuttering steps ("local actions") of the processes is preserved by Two-phase.

Related Work and Our Main Contributions

Partial-order reduction algorithms save computational costs by executing only a subset of the executable transitions from a state, postponing the execution of the rest of the executable transitions without affecting the truth values of the properties being verified. If sufficient care is not taken in the exact method chosen for postponing transitions from processes, there is a danger of indefinitely postponing some of the transitions, which can render the verifier unsound. This is known as the ignoring problem [Pe96].

There have, essentially, been only two partial-order reduction algorithms which have efficient implementations, namely the algorithm used in the SPIN tool [Hol91] and the one used in the PO-PACKAGE tool [God95]. Both these algorithms solve the ignoring problem by using a proviso, first reported in [Pe93, Pe96]. The proviso ensures that the subset of transitions selected at 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 also [HGP92] require that at least one of the transitions do not generate a state in the stack, whereas [HP94] requires the 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 linear time temporal logic (LTL) formulae (safety and liveness), whereas the weaker proviso preserves only stutter free safety properties [HGP92, HP94, Pe93, Pe96]. We observed that in a large number of practical examples arising in our problem domain, such as validation of directory based coherence protocols and server-client protocols, the proviso causes all existing partial order reduction algorithms 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. The algorithm of [God95] 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 Two-phase that finishes comfortably on this protocol and preserves all stutter-free safety properties without using the proviso.

Our contributions in this paper are as follows. We empirically demonstrate on a large number of practical examples that the use of the proviso causes the existing partial order reduction algorithms to be ineffective, often generating far too many states. Our set of examples include the largest of the benchmark examples used by [HP94] and also many large examples arising in our own research (the latter are significantly more involved than the former). We provide intuitive reasons as to
why the use of the proviso can be harmful. We then present our Two-phase algorithm that avoids
the proviso. We prove that the Two-phase algorithm has the same power as the weak-proviso
algorithms—namely that it can establish all stutter-free safety properties. (This proof implies
that the ignoring problem is avoided.) We conclude the paper with a summary of future research
directions.

Organization
Section 2 provides background information about model checking and partial order reduction, aided
by illustrative examples. Section 3 describes the algorithm presented in [HP94] and our own 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 experimental
results on a number of examples. Section 6 provides concluding remarks and plans for future work.

2 Background
The tools SPIN [Hol91] and PO-PACKAGE [God95] as well as our PV 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. 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.

For the sake of simplicity we assume in this paper 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 communi-
cation. This allows us to focus here on the purely interleaved model. (In the actual implementation
of our verifier, Two-phase is modified to handle channels that have multiple senders/multiple receivers
and/or have zero capacity.)

An Intuitive Explanation of Two-phase
To motivate the partial-order reduction examples to be discussed in this paper, consider a con-
current system described in Figure 1. Two processes P and Q communicating via asynchronous
channels (bounded buffers) are shown. It is assumed that variables L and L1 are local (i.e. neither
is accessed by another process nor are they used in any LTL property to be verified). Initially
channel c is assumed to be full and d is neither full nor empty. Because of this assumption, c!msg
is not executable in state <P0, Q0> whereas d?L and c?L1 are both executable in that state. The
set of all possible executions starting from the state <P0, Q0> are as shown. Typically this state
space would be generated by a DFS procedure [Hol91]. Although this procedure is semantically
correct, it can result in redundant interleavings as evidenced by the presence of d?L; c?L1 as well as
c?L1; d?L in the tree. In this example, in order to establish a stutter-free LTL property, it suffices
to generate the execution space within the dotted box as the only state that is outside the dotted
box but not in the box, namely <P2, Q0>, is "covered" by the states <P0, Q0> and <P2, Q1> that are
inside the dotted box. This is what a partial-order reduction algorithm would attempt to do. Thus,
a partial-order reduction algorithm needs to pick only c?L1 from state <P0, Q0>.

Because both d?L; c?L1 and c?L1; d?L are possible from state <P0, Q0>, one may be tempted
to conclude that a partial-order reduction algorithm can pick d?L from <P0, Q0>, ignoring the
execution possibility of c?L1 from <P0,Q0>. However, doing so is unsound for this example. To see this, observe that the d?L successor of <P0,Q0> is only c?L1 where as a c?L1 successor of <P0,Q0> includes both d?L and c!msg. Within the dotted box (and also in the full execution space) we have the possibility of sending "msg" or "L" into the c channel while the execution ensuing from <P2,Q0> allows only the latter possibility. Thus, ignoring the execution possibility of c?L1 from <P0,Q0> doesn’t explore all possible behaviors.

Thus the characterization that c?L1 and d?L can, between themselves, commute is not sufficient to effect partial-order reduction decisions at run-time. Instead, the Two-phase algorithm uses the notion of a deterministic process. Informally, a deterministic process P in state s is one which has (i) exactly one transition executable in s, (ii) the execution of a transition of some other process from s does not affect the executability of the transitions of P in s, and (iii) executing P in s does not disable any transition of other processes. For example, state <P0,Q0> is not deterministic for P because executing Q via c?L1 from <P0,Q0> enables c!msg in the resulting state <P0,Q1>. Note that c!msg was a transition that was disabled before. In other words, the executability of c!msg is affected by the execution of Q. However <P0,Q0> is deterministic for Q.

A partial-order reduction algorithm reduces the number of transitions examined without changing the truth value of the stutter-free LTL formulae being checked. In our Two-phase algorithm, partial-order reductions are effected by the execution of deterministic processes. Notions similar to deterministic have been used in other partial-order reduction works (stubborn sets in [Val93] and ample sets in [Pen96]). We find that the notion of deterministic is efficient to implement and also reduces the number of states searched compared to previous algorithms.
process P
{
    int 1;
    if
        :: c ! 7 -> skip;
        :: d ! 1 -> skip;
    fi;
    l := 0;
    g := 1;
    if
        :: l == 0 -> skip
        :: l != 0 -> skip
    fi;
}

Figure 2: A sample process to illustrate definitions

Terminology Used in Two-phase

We now present the above intuitions more formally, with the aid of Figure 2, where g is a global variable, 1 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. 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.

**local:** A transition (a statement) is said to be *local* if it does not involve a global variable. Examples: c!7, d?1, l:=0, l==0, l!=0, and skip. *local* is a static notion, i.e., a compiler annotates a transition as local.

**global:** A transition is said to be *global* if it involves a global variable. Example: g:=1.

**internal:** A **control state** (program counter) of a process is said to be *internal* if all the transitions possible from it are *local* transitions. This is also a static notion. Example: In Figure 2, the control state corresponding to the entry point of the first if statement is internal since the two transitions possible here, namely c!7 and d?1, are local transitions.

**unconditionally safe:** A *local* transition is said to be *unconditionally safe* if, for all states s, if the transition is *executable* (*non executable*) in s, then it remains *executable* (*non executable*) in state s' resulting from the execution of any sequence of transitions T by other processes from s. *unconditionally safe* is a static notion. Examples: l:=0, l==0, l!=0. In other words, if a transition (e.g. l==0 of P) is executable in a state, then it remains executable in any state attained after the execution of an arbitrary sequence of transitions of other processes from that state. Likewise, if l==0 is disabled in a state, then it remains disabled after the execution of an arbitrary sequence of transitions of other processes from that state.

As an illustration of something that is *not* unconditionally safe, consider the transition d?1. If channel d is empty in a state s, transition d?1 is not executable in s. However, if some other process executes transition d!msg from s resulting in s', transition d?1 becomes executable in s'. Thus, d?1 is *not* unconditionally safe. (However it is conditionally safe, as defined below).
conditionally safe: A conditionally safe transition \( t \) behaves like an unconditionally safe transition in some of the reachable states characterized by a safe execution condition \( p(t) \). More formally, a local transition \( t \) is said to be conditionally safe whenever, in state \( s \in p(t) \), if \( t \) is executable (non executable) in \( s \), then \( t \) is executable (non executable) in state \( s' \) resulting from the execution of any sequence of transitions \( T \) by other processes from \( s \). Conditionally safe is a dynamic notion, i.e., the value of the safe execution condition depends on run-time information such as value of the variables and/or channel contents. Example: \( c!7 \) is a conditionally safe transition. Its safe execution condition is all those states where \( c \) is not full. In such a state \( s \), \( c!7 \) behaves like an unconditionally safe transition for the following reasons: (i) \( c!7 \) is executable in \( s \); (ii) the only effect that a sequence of transitions \( T \) of other processes from \( s \) can have on \( c \) is to consume messages from it (recall that channels are point to point connections). Thus \( c!7 \) remains executable after \( T \).

Another example of a conditionally safe transition is \( d?1 \), with the safe execution condition being that \( d \) be not empty.

safe: A transition \( t \) is safe in a state \( s \) if \( t \) is an unconditionally safe transition or \( t \) is conditionally safe whose safe execution condition is true in \( s \). safe is a dynamic notion, i.e., determining if a transition is safe in a state may require run-time information.

deterministic: A process \( P \) is said to be deterministic in state \( s \), written \( \text{deterministic}(P, s) \), if the control state of \( P \) in \( s \) is internal, if all transitions of \( P \) from this control state are safe, and exactly one transition of \( P \) is executable. deterministic is a dynamic notion. Example: In Figure 2, if control state of \( P \) is at the second if statement, \( P \) is deterministic since only one of the two conditions \( 1==0 \) and \( 1!=0 \) can be true.

The above definition of deterministic formalizes the informal definition provided on page 4.

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 executable at \( S' \). The second phase of executing all transitions of \( S' \) ensures 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 executable transitions of that process. If no such process can be found, all executable 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
initialize stack to contain initial state
initialize cache to contain initial state
proviso() {
    s := top(stack);
    if (found) {
        tr := \{t | t is executable in s and PID(t)=i\};
        nxt := successors of s obtained by executing transitions in tr;
    } else {
        tr := all executable 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);
            proviso();
    }
    pop(stack);
}

initialize stack to contain initial state
initialize cache to \Phi

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

Figure 3: proviso() is a partial order reduction algorithm using weak proviso. TwoPhase() avoids proviso using a different execution strategy.

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 proviso() in Figure 3. In this algorithm, Choose(s) is used to find a process satisfying the above criterion. As mentioned earlier, the proviso (weak or strong) can cause the algorithm to generates many unnecessary states. In some protocols, e.g., Figure 4 (a), all reachable states in the protocol are generated. Figure 4 (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 TwoPhase() in Figure 3. In the first phase, TwoPhase() 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 executable transitions at s are examined.

Note that in the first phase only deterministic processes are executed. An important consequence of deterministic is the following. Suppose P is deterministic in state S. If executing P from S results in state S1, then the set of executable transitions of Pprocesses in S1 is a superset of those executable while in S. Hence, if a transition t of a process Q other than P is executed in state S1,
it is not necessary to execute \( t \) in \( S \), thus saving some work. This fact is proved in Section 4.

### Impact of Proviso on Execution

The Two-phase algorithm outperforms the proviso algorithm and [God95] algorithm when the proviso is invoked often; 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 4, in certain examples, \( \text{proviso}() \) 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. Two-phase does not use the proviso. Instead it alternates one step of partial order reduction step with one step of complete depth first search. Thus 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 Two-phase. 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, Two-phase would not perform well.

### 4 Correctness of the Two-phase Algorithm

We show that \( \text{TwoPhase()} \) preserves all stutter free safety properties. To establish the correctness of \( \text{TwoPhase()} \), we need the following two lemmas.

**Lemma 1:** A state \( X \) is added to \text{cache} only after ensuring that all transitions executable at \( X \) will be executed at \( X \) or at a successor of \( X \). This lemma asserts that \( \text{TwoPhase()} \) does not suffer from ignoring problem.

**Proof:** Proof is based on induction on the “time” a state is entered into \text{cache}.

**Induction Basis:** During the first call of \( \text{TwoPhase()} \) the outer “if” statement of the second phase will be executed; during this phase, all states in \text{list} are added to \text{cache} in the body of the “if” statement. Following that the algorithm examines all successors of \( s \). Let \( s' \) be an arbitrary element of \text{list}. By the manner in which \text{list} is generated, \( s' \) can reach \( s \) via zero or more deterministic transitions. By the definition of deterministic transition, any executable transition 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 \text{cache} satisfies Lemma 1.

**Induction Hypothesis:** Let the states entered into \text{cache} during the first \( i-1 \) calls to \( \text{TwoPhase()} \) be \( s_1, s_2, ..., s_{n-1} \). Assume by induction hypothesis that all executable transitions 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 \text{cache} during the \( i^{th} \) call to \( \text{TwoPhase()} \) 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 \( \text{list} \). Any executable transition at a state \( s' \) and not taken in the path from \( s' \) to \( s \) is also executable at \( s \). Therefore \( s' \) can be added to \( \text{cache} \) without violating the lemma. In the second case, \( s \) is already in \( \text{cache} \); it was entered during an earlier call to \( \text{TwoPhase}() \). By induction hypothesis, all executable transitions of \( s \) are already executed or guaranteed to be executed. Hence all executable transitions of \( s' \) were either already considered at \( s \) or guaranteed to be executed by the hypothesis. Hence adding \( s' \) to \( \text{cache} \) does not violate the lemma. Thus in both cases, \( \text{list} \) can be added to \( \text{cache} \) without violating the lemma.

**Lemma 2:** \( \text{TwoPhase}() \) terminates after a finite number of calls.

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

**Theorem 1:** \( \text{TwoPhase}() \) preserves all stutter-free safety LTL properties.

**Proof:** (Informal) The proof of the theorem follows from the observation that establishing stutter-free safety LTL properties about a finite-state model requires every executable 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 (i.e. stuttering steps do not matter). \( \text{TwoPhase}() \) satisfies these two conditions. In particular, \( \text{TwoPhase}() \) 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 \( \text{TwoPhase}() \) where only deterministic processes are considered (a deterministic process has exactly one executable transition which is also a safe transition). Lemma 1 guarantees that all executable transitions at every state are considered by \( \text{TwoPhase}() \). Hence, \( \text{TwoPhase}() \) preserves the truth-value of stutter-free safety properties.

## 5 Case studies

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

### 5.1 Extremal cases

Figures 4 and 5 show two very simple protocols. Tables 1 and 2 show the results of running the algorithms on these two protocols. For the protocol shown in Figure 4, on a system comprising of \( n \) processes, Two-phase generates \( 2n + 1 \) states where as the proviso algorithm generates \( 3^n \) states. In other words, Two-phase reduces the time and space requirements by an exponential factor compared to the proviso algorithm. The reason for better performance of Two-phase is that the initial state is reached many times during the DFS analysis by the proviso algorithm. Since the initial state is
always in the stack, the proviso is invoked many times, thus increasing the number of states visited by the algorithm. On the other hand, for the protocol in Figure 5, on a system comprising of \( n \) processes, the Two-phase algorithm generates \( 3^n \) states where as the proviso algorithm generates \( 2^{(n+1)} - 1 \) states, i.e., Two-phase incurs an exponential penalty compared to the proviso algorithm. The reason for the poor performance of Two-phase algorithm on this protocol is that none of the reachable states is deterministic with respect to any process. Hence, Two-phase degenerates to classical depth first search.

<table>
<thead>
<tr>
<th>( N )</th>
<th>Proviso Algorithm</th>
<th>[God95] Algorithm</th>
<th>Two-phase Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>81/0.32</td>
<td>70/0.35</td>
<td>9/0.33</td>
</tr>
<tr>
<td>5</td>
<td>243/0.34</td>
<td>217/0.42</td>
<td>11/0.33</td>
</tr>
<tr>
<td>6</td>
<td>729/0.38</td>
<td>683/0.64</td>
<td>13/0.33</td>
</tr>
<tr>
<td>7</td>
<td>2187/0.50</td>
<td>2113/1.4</td>
<td>15/0.33</td>
</tr>
<tr>
<td>8</td>
<td>6561/0.83</td>
<td>6422/4.34</td>
<td>17/0.33</td>
</tr>
</tbody>
</table>

Table 1: Number of states saved in the hash table, and time taken by different algorithms on the Best Case.
Figure 5: Worst case protocol. Statistics for this protocol are in Table 2.

<table>
<thead>
<tr>
<th>N</th>
<th>Proviso Algorithm</th>
<th>[God95] Algorithm</th>
<th>Two-phase Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>63/0.33</td>
<td>64/0.37</td>
<td>243/0.39</td>
</tr>
<tr>
<td>6</td>
<td>127/0.39</td>
<td>128/0.42</td>
<td>729/0.49</td>
</tr>
<tr>
<td>7</td>
<td>255/0.43</td>
<td>256/0.51</td>
<td>2187/0.76</td>
</tr>
<tr>
<td>8</td>
<td>511/0.43</td>
<td>512/0.7</td>
<td>6561/1.71</td>
</tr>
<tr>
<td>9</td>
<td>1023/0.51</td>
<td>1024/1.21</td>
<td>19683/4.88</td>
</tr>
</tbody>
</table>

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

5.2 Wavefront Arbiter

A cross-bar arbiter that operates by sweeping diagonally propagating “wavefronts” within a circuit array [Gop94] is shown in Figure 6. 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.

<table>
<thead>
<tr>
<th>N</th>
<th>Proviso Algorithm</th>
<th>Two-phase Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>281/0.53</td>
<td>172/0.42</td>
</tr>
<tr>
<td>7</td>
<td>384/0.68</td>
<td>230/0.47</td>
</tr>
<tr>
<td>8</td>
<td>503/0.91</td>
<td>296/0.56</td>
</tr>
<tr>
<td>9</td>
<td>638/1.30</td>
<td>370/0.69</td>
</tr>
<tr>
<td>10</td>
<td>789/1.76</td>
<td>452/0.89</td>
</tr>
</tbody>
</table>

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 due to implementation limitations of the algorithm.
Figure 6: 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.

5.3 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. The migratory protocol contains about 200 lines of Promela code and the 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 the invalidate protocol. This algorithm aborted search after generating more than 963,000 states due to unavailability of more memory. In contrast, Two-phase 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.
<table>
<thead>
<tr>
<th>Protocol</th>
<th>Proviso Algorithm</th>
<th>[God95] Algorithm</th>
<th>Two-phase Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Migratory</td>
<td>34906/5.08</td>
<td>28826/14.45</td>
<td>23163/2.84</td>
</tr>
<tr>
<td>Invalidate</td>
<td>Unfinished</td>
<td>Unfinished</td>
<td>193389/19.23</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>N</th>
<th>Proviso Algorithm</th>
<th>[God95] Algorithm</th>
<th>Two-phase Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>295/0.47</td>
<td>242/0.47</td>
<td>272/0.34</td>
</tr>
<tr>
<td>3</td>
<td>11186/3.43</td>
<td>8639/7.74</td>
<td>3232/0.83</td>
</tr>
<tr>
<td>4</td>
<td>Unfinished</td>
<td>Unfinished</td>
<td>62025/14.9</td>
</tr>
</tbody>
</table>

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

5.4 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 this 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. The 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.

5.5 Other Protocols

We also ran Two-phase on the protocols provided as part of the SPIN distribution. Some of the protocols supplied with the SPIN distribution are not perpetual processes (i.e., they terminate or deadlock). The 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 the same 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 Two-phase 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 a new algorithm called Two-phase for partial order reduction that preserves stutter-free safety properties. Unlike the proviso algorithm or [God95] algorithm, Two-phase does not use the proviso. Instead it alternates one step of partial order reduction step (using deterministic
<table>
<thead>
<tr>
<th>Protocol</th>
<th>Proviso Algorithm</th>
<th>[God95] Algorithm</th>
<th>Two-phase Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sort</td>
<td>174/0.35</td>
<td>173/0.6</td>
<td>174/0.33</td>
</tr>
<tr>
<td>Snoopy</td>
<td>20323/6.22</td>
<td>10311/10.53</td>
<td>20186/5.08</td>
</tr>
<tr>
<td>Pftp</td>
<td>161751/34.5</td>
<td>125877/150.7</td>
<td>230862/36.3</td>
</tr>
</tbody>
</table>

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.

transitions) followed by one step of classical depth first search (using all transitions). This algorithm is shown to perform better than the other existing 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, Two-phase algorithm may generate more states than the algorithms using proviso.

It is possible to modify the first phase of Two-phase to make use of all safe transitions instead of using just deterministic transitions. The advantage of such an algorithm would be that, unlike Two-phase that degenerates 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 more complex [Sta]) Also, [HP94] preserves all stutter free LTL formulae [Pel93, Pel96]. We proved that Two-phase algorithm preserves only stutter free safety properties. It is not difficult to see that Two-phase liveness properties also. [Liv] presents a proof that Two-phase preserves liveness properties in a limited setting where all transitions are either unconditionally safe or unsafe, i.e., conditionally safe transitions are not present.
References


