A Partial Order Reduction Algorithm without the Proviso

Ratan Nalumasu
Ganesh Gopalakrishnan

UUCS-98-017

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

August 19, 1998

Abstract

This paper presents a partial order reduction algorithm, called Two phase, that preserves stutter free LTL properties. Two phase dramatically reduces the number of states visited compared to previous partial order reduction algorithms on most practical protocols. The reason can be traced to a step of the previous algorithms, called the *proviso* step, that specifies a condition on how a state that closes a loop is expanded. Two phase avoids this step, and uses a new execution approach to obtain the reductions. Two phase can be easily combined with an on-the-fly model-checking algorithm to reduce the memory requirements further. Furthermore a simple but powerful selective-caching scheme can also be added to Two phase. Two phase has been implemented in a model-checker called PV (Protocol Verifier) and is in routine use on large problems.

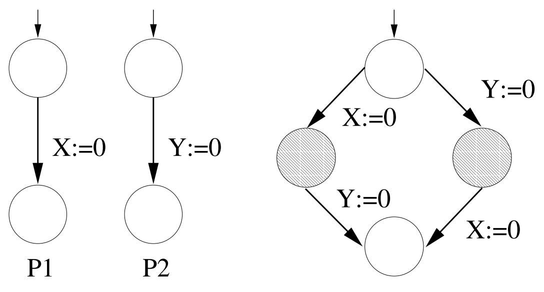
Keywords: Partial order reductions, explicit enumeration, temporal logic, on-the-fly model-checking, concurrent protocol verification

1 Introduction

With the increasing scale of software and hardware systems and the corresponding increase in the number and complexity of concurrent protocols involved in their design, formal verification of concurrent protocols is an important practical need. Automatic verification of finite state systems based on explicit state enumeration methods [CES86, Hol91, Dil96, HP96] have shown considerable promise in real-world protocol verification problems and have been used with success on many industrial designs [Hol97, DPN93]. Using most explicit state enumeration tools, a protocol is modeled as a set of concurrent processes communicating via shared variables and/or communication channels [HP96, Dil96]. The tool generates the state graph represented by the protocol and checks for the desired temporal properties on that graph. A common problem with this approach is that state graphs of most practical protocols are quite large, and the size of the graph can often increase exponentially with the size of the protocol, commonly referred to as *state explosion*.

The interleaving model of execution used by these tools is one of the major causes of state explosion. This is shown through a simple example in Figure 1. Figure 1(a) shows a system with two processes P1 and P2, and Figure 1(b) shows the state space of this example. If the property under consideration does not involve at least one of the variables X and Y, then one of the two shaded states need not be generated, thus saving one state. A straightforward extension of this example to n processes would reveal that an interleaving model of execution would generate 2^n states where $n+1$ would suffice.

Partial order reductions attempt to bring such reductions by exploiting the fact that in realistic protocols, there are many transitions that "commute" with each other, and hence it is sufficient to explore those transitions in any *one order* to preserve the truth value of the temporal property under consideration. In essence, from every state, a partial order reduction algorithm selects a *subset* of transitions to explore, while a normal graph traversal such as depth first search (DFS) algorithm would explore all transitions. Partial order reduction algorithms play a very important role in mitigating state explosion, often reducing the computational and memory cost by an exponential factor.



(a) Simple system

(b) State graph

Figure 1: (b) shows the state graph of P1 and P2 in (a).

This paper presents a new partial order reduction algorithm called Two phase, that in most practical cases outperforms existing implementations of the partial order reductions. The algorithm is implemented in a tool called PV ("Protocol Verifier") that finds routine application in our research.

To our knowledge, so far there have been only two partial order reduction algorithms which have implementations: the algorithm presented in [Pel96, HP94] and [God95]. The [Pel96, HP94] algorithm is implemented in the explicit state enumeration model-checker SPIN, and also in implicit state exploration tools VIS and COSPAN [ABHQ97, KLM+97, NK95]. The [God95] algorithm is implemented in PO-PACKAGE tool. Both these algorithms solve the ignoring problem by using *provisos*, whose need was first recognized by Valmari [Val92]. Provisos ensures that the subset of transitions selected at a state do not generate a state that is in the stack maintained by the 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. The [God95] algorithm (and also the [HGP92] algorithm) require that at least one of the selected transitions do not generate a state in the stack, where as [HP94] requires the stronger condition that no selected transition generates a state in the stack. The stronger proviso is sufficient to preserve all stutter free linear time temporal logic (LTL-X) formulae (safety and liveness), where as the weaker proviso preserves only stutter free safety properties [HGP92, HP94, Pel93, Pel96]. 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 provisos cause 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 270,000 states when limited to 64MB memory usage. 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 the fact that our partial order reduction algorithm, Two phase, that does not use the proviso, finishes comfortably on this protocol. In fact, as showed in Section 6, in all non-trivial examples, Two phase outperforms proviso based algorithms. Two phase is implemented in a model-checker called PV ("Protocol Verifier"). The tool can be obtained by contacting the authors.

The first major difference between Two phase and other partial order reduction algorithms is the way the algorithms expand a given state. Other partial order reduction algorithms attempt to expand each state visited during the search using a subset of enabled transitions at that state. To address the ignoring problem, the algorithms use a proviso (or a condition very similar to the provisos). Two phase search strategy is completely different: when it encounters a new state x, it first expands the state using only *deterministic* transitions in its first phase resulting in a state y. (Informally, deterministic transitions are the transitions that can be taken at the state without effecting the truth property of the property being verified.) Then in the second phase, y is expanded *completely*. The advantage of this search strategy is that it is not necessary to use a proviso. As the results in Section 6 show, this often results in a much smaller graph.

The second major difference is that Two phase naturally supports *selective caching in conjunction* *with on-the-fly model-checking*. An explicit enumeration search algorithm typically saves the list of visited states in a hash table ("cached"). Since the number of visited states is large, it would be beneficial if not all visited states need to be stored in the hash table, referred to as selective caching. On-the-fly model-checking means that the algorithm finds if the property is true or not as the state graph of the system is being constructed (as opposed to finding only after the graph is completely constructed). It is difficult to combine the on-the-fly model-checking algorithm, partial-order reductions, and selective-caching together due to the need to share information among these three aspects. [HPY96] showed that previous attempts at combining proviso based algorithms with the on-the-fly algorithm of [CVWY90] have been erroneous. However, thanks to the simplicity of the first phase of Two phase algorithm, it can be combined easily with the on-the-fly algorithm of [CVWY90]. Also Two phase lends itself to be used in conjunction with a simple but effective selective-caching strategy, and its correctness is very easy to establish.

To summarize, the contributions of this paper are:

1. A new partial order reduction algorithm called Two phase that does not use the proviso,
2. A proof of correctness of Two phase,
3. A selective caching scheme that can be quite naturally integrated with Two phase, and
4. An evaluation of performance of the algorithm compared to other implementations using the PV model-checker.

The rest of the paper is organized as follows. Section 2 presents definitions and background. Section 3 presents the basic depth first search algorithm, the partial order reduction algorithm of [Pel96] (algorithms in [Val92, HP94, God95] are very similar), and Two phase, as well as a proof that Two phase preserves all LTL-X properties. Section 5 presents the on-the-fly model-checking algorithm of [CVWY90], and discusses on how it can be combined with Two phase. This section also presents a selective caching strategy and shows how it can be combined with Two phase. Section 6 compares the performance of [Pel96] algorithm (implemented in SPIN) with that of Two phase. (implemented in PV), and provides a qualitative explanation of the results. Finally, Section 7 provides concluding remarks.

Related work

Lipton [Lip75] suggested a technique to avoid exploring the entire state graph to find if a concurrent system deadlocks. Lipton notes that execution of some transitions can be postponed as much as possible (*right movers*) and some transitions can be executed as soon as possible (*left movers*) without affecting the deadlocks. Partial order reductions can be considered as a *generalization* of this idea to verify richer properties than just deadlocks.

Valmari [Val92, Val93] has presented a technique based on *stubborn sets* to construct a reduced graph to preserve the truth value of all stutter free LTL formulae. The [Val92] algorithm uses a general version of the proviso mentioned above. The [Val93] algorithm does not use the proviso, but avoids the ignoring problem by choosing "large stubborn sets". [GW92, GP93, God95] present a partial order theory based on traces to preserve safety properties, using a slight variation of the proviso, implemented in PO-PACKAGE. [Pel96] presents a partial order reduction algorithms based on *ample* sets and the strong proviso. [HP94] presents an algorithm very similar to and based on the algorithm of [Pel96], implemented in SPIN [HP96]. The [Pel96] algorithm is discussed in Section 3. Since the implementations of the two algorithms are similar, whenever one algorithm fails to bring much reductions, so does the other.

The version of the proviso discussed earlier, first appeared in [Pel93], is shown to be sufficient to preserve all liveness properties. In [Val92] a more general condition for correctness is given: if (a) every elementary loop in the reduced graph contains at least one state where all global transitions (visible transitions in their terminology) are expanded, (b) at every state s, if there is an enabled local transition, then the set of transitions chosen to be expanded at s contains at least one local transition then the reduced graph preserves all LTL-X formulae on global transitions. Two phase does not use the provisos, and instead uses *deterministic* transitions to bring the reductions. Two phase has been previously reported in [NG97b, NG97a, NG96].

2 Definitions and Notation

We assume a process oriented modeling language with each process maintaining a set of local variables that only it can access. The value of these local variables form the *local state* of the process. For convenience, we assume that each process maintains a distinguished local variable called program counter ("control state"). A concurrent system or simply system consists of a set of processes, a set of global variables and point-to-point channels of finite capacity to facilitate communication among the processes. The global state, or simply the "state" of the system consists of local states of all the processes, values of the global variables, and the contents of the channels. $\mathcal{S}$ denotes the set of all possible states ("syntactic state") of the system, obtained simply by taking the Cartesian product of the range of all variables (local variables, global variables, program counters, and the channels) in the system. The range of all variables (local, global, and channels) is assumed to be finite, hence $\mathcal{S}$ is also finite.

Each program counter of a process is associated with a finite number of transitions. A transition of a process P can read/write the local variables of P, read/write the global variables, send a message on the channel on which it is a sender, and/or receive a message from the channel for which it is a receiver. A transition may not be enabled in some states (for example, a receive action on a channel is enabled only when the channel is non-empty). If a transition t is enabled in a state $s \in \mathcal{S}$, then it is uniquely defined. Non-determinism can be simulated simply by having multiple transitions from a given program counter. We use $t, t^{'}$ to indicate a transition, $s \in \mathcal{S}$ to indicate a state in the system,

$t(s)$ to indicate the state that results when t is executed from s, P to indicate a sequential process in the system, and pc(s,P) to indicate the program counter (control state) of P in s, and pc(t) to indicate the program counter with which the transition t is associated.

local: A transition (a statement) is said to be *local* if it does not involve any global variable.

global: A transition is said to be *global* if it involves one or more global variables. Two global transitions of two different processes may or may not commute, while two local transitions of two different processes commute.

internal: A control state (program counter) of a process is said to be *internal* if all the transitions associated with it are *local* transitions.

unconditionally safe: A *local* transition t is said to be *unconditionally safe* if, for all states $s \in \mathcal{S}$, if t is enabled (disabled) in $s \in \mathcal{S}$, then it remains enabled (disabled) in $t^{'}(s)$ where $t^{'}$ is any transition from another process. Note that if t is an unconditionally safe transition, by definition it is also a *local* transition. From this observation, it follows that executing $t^{'}$ and t in either order would yield the same state, i.e., t and $t^{'}$ commute. This property of commutativity forms the basis of the partial order reduction theories.

Note that channel communication statements are *not unconditionally safe*: if a transition t in process P attempts to read, and the channel is empty, then the transition is disabled; however, when a process Q writes to that channel, t becomes enabled. Similarly, if a transition t of process P attempts to send a message through a channel, and the channel is full, then t is disabled; when a process Q consumes a message from the channel, t becomes enabled.

conditionally safe: A *conditionally safe* transition t behaves like an *unconditionally safe* transition in some of the states characterized by a *safe execution condition* $p(t) \subseteq \mathcal{S}$. More formally, a local transition t of process P is said to be *conditionally safe* whenever, in state $s \in p(t)$, if t is enabled (disabled) in s, then t is also enabled (disabled) in $t^{'}(s)$ where $t^{'}$ is a transition of process other than P. In other words, t and $t^{'}$ commute in states represented by $p(t)$.

Channel communication primitives are *conditionally safe*. If t is a receive operation on channel c, then its safe execution condition is "c is not empty". Similarly, if t is a send operation on channel c, then its safe execution condition is "c is not full".

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, i.e., $s \in p(t)$.

deterministic: A process P is said to be *deterministic* in s, written *deterministic(P, s)*, if the control state of P in s is *internal*, all transitions of P from this control state are *safe*, and exactly one transition of P is enabled.

independent: Two transitions t and $t^{'}$ are said to be independent of each other iff at least one of the transitions is *local*, and they belong to different processes.

The partial order reduction algorithms such as [Val92, Pel96, HP94, God95] use the notion of *ample set* based on *safe* transitions. Our Two phase algorithm uses the notion of *deterministic* to bring reductions. The proof of correctness of Two phase algorithm uses the notion of *independent* transitions.

2.1 Linear temporal logic and Büchii automaton

A LTL-X formulae is a LTL formulae without the next time operator X. Formally, system LTL-X (*linear-time logic without next time operator* or stutter free LTL) is defined from atomic propositions $p_1 \dots p_n$ by means of boolean connectives, $\Box$ ("always"), $\Diamond$ ("eventually"), and U ("until") operators. If $\alpha = \alpha(0) \dots \alpha(\omega)$ is an infinite sequence of states that assign a truth value to $p_1 \dots p_n$, ϕ a LTL-X formulae, then the satisfaction relation $\alpha \models \phi$ is defined as follows:

$$
\begin{array}{lll}
\alpha \models p_i & \text{iff} & \alpha(0) \models p_i \\
\alpha \models \phi_1 \wedge \phi_j & \text{iff} & \alpha \models \phi_1 \text{ and } \alpha \models \phi_2 \\
\alpha \models \neg\phi & \text{iff} & \neg(\alpha \models \phi) \\
\alpha \models \Box\phi & \text{iff} & \forall i \geq 0 : \ \alpha(i) \dots \alpha(\omega) \models \phi \\
\alpha \models \Diamond\phi & \text{iff} & \exists i \geq 0 : \ \alpha(i) \dots \alpha(\omega) \models \phi \\
\alpha \models \phi_1 U \phi_2 & \text{iff} & \exists i \geq 0 : \ \alpha(i) \dots \alpha(\omega) \models \phi_2 \\
 & & \text{and } \forall 0 \leq j < i : \ \alpha(j) \dots \alpha(\omega) \models \phi_1
\end{array}
$$

If M is a concurrent system, then $M \models \phi$ is true iff for each sequence α generated by M from the initial state, $\alpha \models \phi$.

Büchii automaton [vL90] are non-deterministic finite automata with an acceptance condition to specify which infinite word (ω-word) is accepted by the automaton. Formally, a Büchii automaton is a tuple $A = (Q, q_0, \Sigma, \Delta, F)$ where Q is the set of the states, q_0 is the initial state, Σ is the input, $\Delta \subseteq Q \times \Sigma \times Q$, and $F \subseteq Q$ is the set of final states. A *run* of A on an ω-word $\alpha = \alpha(0)\alpha(1)\dots$ from Σ^ω is an infinite sequence of states $\sigma = \sigma(0)\sigma(1)\dots$ such that $\sigma(0) = q_0$ and $(\sigma(i), \alpha(i), \sigma(i+1)) \in \Delta$. The sequence α is accepted by A iff at least one state of F appears infinitely often in σ.

The model-checking problem, $M \models \phi$, may be viewed as an *automata-theoretic verification* problem, $L(M) \subseteq L(\phi)$ where $L(M)$ and $L(\phi)$ are languages accepted by M and the linear-time temporal formulae ϕ respectively. If an ω automaton such as the Büchii automaton $A_{\neg\phi}$ accepts the language $\overline{L(\phi)}$, the verification problem of $L(M) \subseteq L(\phi)$ can be answered by constructing the state graph of the synchronous product of M and $A_{\neg\phi}$, $S = M \otimes A_{\neg\phi}$. If any strongly connected components of the graph represented by S satisfies the acceptance condition of $A_{\neg\phi}$ then and only then ϕ is violated in M [Kur94].

```
model_check()
{
    /* No states are visited */
    V_f := φ;
    /* No edges are visited */
    E_f := φ;
    dfs(InitialState);
}
```

```
dfs(s)
{
    V_f := V_f + {s};
    [1] foreach enabled transition t in s do
        [2] E_f := E_f + {(s,t,t(s)};
        if t(s) ∉ V_f then
            dfs(t(s))
        endif
    endforeach
}
```

Figure 2: Basic depth first search algorithm

3 Basic DFS and Proviso Based Partial Order Reduction Algorithms

Figure 2 shows the basic depth first search (DFS) algorithm used to construct the full state graph a protocols. V_f is a hash table ("visited") used to cache all the states that are already visited. Statement $\boxed{1}$ shows that the algorithm expands *all* transitions from a given state. Statement $\boxed{2}$ shows how the algorithm constructs the state graph of the system in E_f.

Partial order reduction based search algorithms attempt to replace $\boxed{1}$ by choosing a subset of transitions. The idea is that if two transitions t and $t^{'}$ commute with each other in a state s and if the property to be verified is insensitive to the execution order of t and $t^{'}$, then the algorithm can explore $t(s)$, postponing examination of $t^{'}$ to $t(s)$. Of course, care must be exercised to ensure that no transition is postponed forever, commonly referred to as the *ignoring problem*. The essential nature of these algorithms is shown as `dfs_po` Figure 3. This algorithm also uses `ample(s)` to select a subset of transitions to expand at each step. When `ample(s)` returns a proper subset of enabled transitions, the following conditions must hold: (a) the set of transitions returned commute with all other transitions, (b) none of the transitions result in a state that is currently being explored (as indicated by its presence in `redset` variable maintained by `dfs_po`).

The intuitive reasoning behind the condition (b) is that, if two states s and $s^{'}$ can reach each other, then without this condition s might delegate expansion of a transition to $s^{'}$ and vice versa, hence never exploring that transition at all. Condition (b), sometimes referred to as *reduction proviso* or simply *proviso*, is enforced by the highlighted line in `ample(s)`. If a transition, say t, is postponed at s, then it must be examined at a successor of s to avoid the ignoring problem. However, if $t(s)$ itself being explored (i.e., $t(s)$ ∈`redset`), then a circularity results if $t(s)$ might have postponed t. To break the circularity, `ample(s)` ensures that $t(s)$ is not in `redset`. As we shall see in Section 5.1, the dependency of `ample` on `redset` to evaluate the set of transitions has some very important consequences when on-the-fly model-checking algorithms are used.

```
dfs_po(s)
{
    /* Record the fact that s is partly
       expanded in redset */
    redset := redset + {s};
    V_r := V_r + {s};
    /* ample(s) uses redset */
    1 foreach transition t
            in ample(s) do
        2 E_r:=E_r+{(s,t,t(s)};
        if t(s) ∉ V_r then
            dfs_po(t(s));
        endif;
    endforeach;
    /* s is completely expanded.  So
       remove it from redset */
    redset := redset-{s};
}
```

```
ample(s)
{
  for each process P do
    acceptable := true;
    T := all transitions t of P
      such that pc(t) = pc(s,P);
    foreach t in T do
      if(t is global) or
        (t is enabled and
          (t(s) ∈ redset)) or
        (t is conditionally safe
           and s ∉ p(t)) then
          acceptable := false;
      endif
    endforeach;
    if acceptable and T has at least
          one enabled transition
      return enabled transitions in T;
    endif;
  endforeach;

  /* No acceptable subset of
     transitions is found */
  return all enabled transitions;
}
```

Figure 3: Proviso based partial order reduction algorithm

3.1 Efficacy of partial order reductions

The partial order reduction algorithm shown in Figure 3 can reduce the number of states by an exponential factor [HP94, Pel96]. However, in many practical protocols, the reductions are not as effective as they can be. The reason can be traced to the use of proviso. This is motivated using the system shown in Figure 4. Figure 4(a) shows a system consisting of two sequential processes P1 and P2 that do not communicate at all, i.e., $\tau_1 \ldots \tau_4$ commute with $\tau_5 \ldots \tau_8$. The total number of states in this system is 9. The optimal reduced graph for this system contains 5 states, shown in Figure 4(b).

Figure 4(c) shows the state graph generated by the partial order reduction algorithm of Figure 3. This graph is obtained as follows. The initial state is <s0,s0>. ample(<s0,s0>) may return either $\{\tau_1, \tau_3\}$ or $\{\tau_5, \tau_7\}$. Without loss of generality, assume that it returns $\{\tau_1, \tau_3\}$, resulting in states <s1,s0> and <s2,s0>. Again, without loss of generality, assume that the algorithm chooses to expand <s1,s0> first, where transitions $\{\tau_2\}$ of P_1 and $\{\tau_5, \tau_7\}$ of P_2 are enabled. τ_2(<s1,s0>) = <s0,s0>, and when dfs_po(<s1,s0>) is called, `redset`={<s0,s0>}. As a result ample(<s1,s0>) cannot return $\{\tau_2\}$; it returns $\{\tau_5, \tau_7\}$. Executing τ_5 from (<s1,s0>) results in <s1,s1>, the third state in the figure. Continuing this way, the graph shown in Figure 4(c) is obtained. Note that this system contains all 9 reachable states in the system, thus showing that a proviso based partial order reduction algorithm might fail to bring appreciable reductions. As confirmed by the examples in

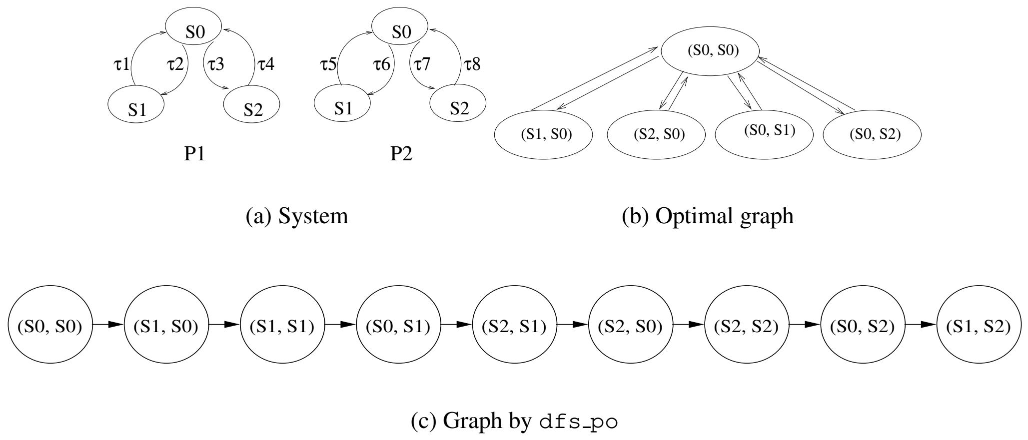


Figure 4: A trivial system, and its optimal reduced graph, and the reduced graph generated by `dfs_po`

Section 6, the algorithm may not bring much reductions in realistic protocols also.

4 The Two Phase Algorithm

As the previous contrived example, the size of the reduced graph generated by a proviso based algorithm can be quite high. This is true even for realistic reactive systems. 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 already demonstrated in Figure 4. Section 6 will demonstrate that in realistic systems also the number of extra states generated due to the proviso can be high.

We propose a new algorithm, Two phase (Figure 5), that does not use the proviso, thus avoiding generating the extra states. In the first phase (`phase1`), `Twophase` executes deterministic processes resulting in a state `s`. In the second phase, *all* enabled transitions at `s` are examined. Two phase algorithm outperforms `dfs_po` (and PO-PACKAGE) when the proviso is invoked often; confirmed by the examples in Section 6. Note that `phase1` is *more general* than coercening of actions. In coercening of actions, two or more actions of a given process are combined together to form a larger "atomic" operation. In `phase1`, actions of multiple processes are executed.

```
model_check()
{
  V_r := φ;
  E_r := φ;
  /* fe (fully expanded) is used in proof */
  fe := φ;
  Twophase();
}
```

```
phase1(in)
{
  s := in;
  list := {s};
  path := {};
  foreach process P do
    while (deterministic(s, P))
      /* Let t be the only enabled
         transition in P */
      olds := s;
      s := t(olds);
      path := path + {(olds, t, s)};
      if (s ∈ list)
         goto NEXT_PROC;
      endif
      [1] list :=  list + {s};
      endwhile;
    NEXT_PROC: /* next process */
  endforeach;
  return(path, s);
}
```

```
Twophase(s)
{
    /* Phase 1 */
    (path, s) := phase1(s);

    /* Phase 2: Classic DFS */
    if s∉V_r then
      /* fe is used in proof */
      [1] V_r := V_r + all states in path;
      [2] E_r := E_r + path;
      [3] fe := fe + {s};
      foreach enabled transition t do
        [3] E_r := E_r + (s, t, t(s));
        if t(s) ∉ V_r then
          Twophase(t(s));
        endif;
      endforeach;
    else
      [1'] V_r := V_r + all states in path;
      [2'] E_r := E_r + path;
    endif;
}
```

Figure 5: Two phase algorithm

4.1 Correctness of Two phase algorithm

We show that the graph generate by `Twophase`, G_r=(V_r, E_r), satisfies a LTL-X property ϕ iff the graph generated by `dfs`, G_f=(V_f, E_f), also satisfies ϕ.

Lemma 1 (Termination) All calls made to `phase1` and `Twophase` terminate.

Proof: In `phase1`, a new state is added to `list` every time the `while` loop is executed. Since the number of states in the system is finite, the loop terminates, hence so does the `phase1`. Similarly, at least one new state is added to V_r every time `Twophase` is called recursively. Hence these calls also terminate. □

Notation: If G=(V,E) is a graph, then a sequence of G is of the form $s_1 \xrightarrow{t_1} s_2 \xrightarrow{t_2} s_3 \ldots$, where each s_i is in V, and $(s_i, t_i s_{i+1})$ is in E. A sequence may be finite or infinite. σ, ρ, σ_1, ρ_1 etc. are used to denote sequences. If $\sigma = s_1 \xrightarrow{t_1} \ldots s_i \xrightarrow{t_i} \ldots s_j \ldots$ is a sequence in G, $\sigma(i) \ldots \sigma(j)$ indicates the subsequence $s_i \xrightarrow{t_i} s_{i+1} \ldots s_j$, and $\sigma(i) \ldots \sigma(\inf)$ indicates the subsequence of σ starting from s_i till the end of σ (if σ is finite, this is equivalent to $s_i \xrightarrow{t_i} s_{i+1} \ldots s_n \xrightarrow{t_n} s_{n+1}$ where $s_n \xrightarrow{t_n} s_{n+1}$ is the last transition of σ). $\Box$

From the construction, it is clear that G_r is a subgraph of G_f. Hence all paths in G_r are also paths in G_f, hence if G_f satisfies ϕ so does G_r. Now we show that if G_f violates ϕ, then so does G_r. Let σ be a path in G_f starting from the initial state that reveals the violation of ϕ. The construction below shows how to transform σ successively obtaining "equivalent" sequences $\sigma_1, \ldots \sigma_n = \rho$, where ρ is a sequence of transitions in G_r that shows the violation of ϕ. To do so, first we need to establish that from every state $x \in V_r$, there is a path to a state $y \in V_r$ where y is completely expanded. Note that when a state y is completely expanded, `Twophase` adds y to `fe` on line $\boxed{3}$.

Lemma 2 (ReachFE) If x is a state in V_r, then there is a finite sequence of $\Pi_x \in G_r$, of length zero or more such that σ takes x to a state $y \in$ `fe`. In addition, $(s, t, s^{'})$ is a transition in Π_x where t belongs to process P, then P is deterministic in s.

Proof: The proof is by constructing Π_x that satisfies the lemma. x is added to V_r either on line $\boxed{1}$ or $\boxed{1'}$ in `Twophase`. We show that the lemma holds by a simple induction on the order in which the states are added to V_r.

Induction basis: During the first call of `Twophase`, V_r is empty; hence the *then* clause of the outermost *if* statement is executed. At this time, all states in `path` are added to V_r, and `s` is completely expanded by the *foreach* statement. x is a state in `path`, the lemma holds with $y =$`s`, with σ being a sub path of `path` starting from x.

Induction hypothesis: Assume that the lemma holds for states added to V_r during the first i calls of `Twophase`.

Induction Step: x is added to V_r in $i + 1$ th call of `Twophase`. There are two cases to consider:
Case i: x is added to V_r on $\boxed{1}$. This case is similar to the induction basis: the lemma holds with $y =$`s` and σ is a sub path of `path` from x to `s`.
Case ii: x is added to V_r on $\boxed{1'}$ (in the *else* clause). In this case `s` is already in V_r. By induction hypothesis, there is a finite sequence, $\sigma^{'}$ from `s` to y where y is in `fe`. Let p be the sub path of `path` from x to `s`. The lemma holds with σ being concatenation of p and $\sigma^{'}$. $\Box$

Note 1 If $\sigma = s_1 \xrightarrow{t_1} s_2 \ldots$ is a (finite or infinite) sequence in G_f, l is a *local* transition of process P, no transitions of P are in σ, and l is enabled at s_1, then $\sigma^{'} = s_1 \xrightarrow{l} {s_1}^{'} \xrightarrow{t_1} s_2^{'} \ldots$ is a sequence in G_f obtained by prepending l to σ, and σ and $\sigma^{'}$ satisfy the same set of LTL-X formulae on the *global* transitions.

Note 2 If σ and $\sigma^{'}$ are two sequences in G_f starting from x and the sequence of transitions in $\sigma^{'}$ is a permutation of the sequence of transitions in σ such that only consecutive independent transitions are reordered, then σ and $\sigma^{'}$ satisfy the same set of LTL-X formulae on the *global* transitions.

Lemma 3 Let $p = s_1 \xrightarrow{t_1} s_2 \xrightarrow{t_2} \ldots s_m \xrightarrow{t_m} s_{m+1} \ldots s_n$ be a subsequence of Π_x for some $x \in V_r$, and t_m be the first transition of some process P in p. If ρ a sequence in G_f starting from s_1 and does not contain t_m then ρ contains *no* transitions from P. (This implies that t_m is independent of all transitions in ρ).

Proof: Assume that the lemma is false, i.e., ρ contains a transition u of P such that $u \neq t_m$. We show that this leads to a contradiction. From the assumptions that $s_m \xrightarrow{t_m} s_{m+1}$ is in Π_x, t_m and u belong to the same process, u is executed in ρ it is clear that:

O1 u and t_m are safe at s_m

O2 t_m is enabled at s_m, and u is disabled at s_m,

O3 u continues to be disabled from every state in a sequence starting from s_m until at least t_i is executed,

O4 u is executed in ρ after some finite number of transitions, and

O5 none of the transitions in $t_1 \ldots t_{m-1}$ belong to P.

We transform an initial segment of $\rho_0 = \rho$ successively into $\rho_1, \rho_2 \ldots \rho_{m-1}$ such that

C1 p and ρ_i are identical for the first i transitions, and

C2 if u is executed at some state in ρ_i then it is also executed at some state (possibly different) in ρ_{i+1}.

By construction ρ_i and Π_x are identical up to the first i transitions. Now we construct ρ_{i+1} from ρ_i such that ρ_{i+1} and Π_x are identical up to first $i+1$ transitions and if u is executed in some state in ρ_i then it is also executed in some state in ρ_{i+1}. Finally we show that u is not executed in all state of ρ_{m-1}, which implies that it is not executed in any state of $\rho_0 = \rho$, leading to a contradiction with (**O4**) above. There are two cases to consider.
Case 1: (Figure 6) t_{i+1} does not appear in ρ_i at all. ρ_{i+1} is constructed by simply inserting t_{i+1} at the appropriate position as shown in the Figure 6. If u is in ρ_i then it will also be in ρ_{i+1}.
Case 2: (Figure 7) t_{i+1} appears in ρ_i. ρ_{i+1} is obtained by moving t_{i+1} such that it is executed from s_{i+1}. (Since t_{i+1} is a *local* transition and is enabled at s_{i+1}, this reordering is allowed.) If u is in ρ_i, then it is also in ρ_{i+1}.

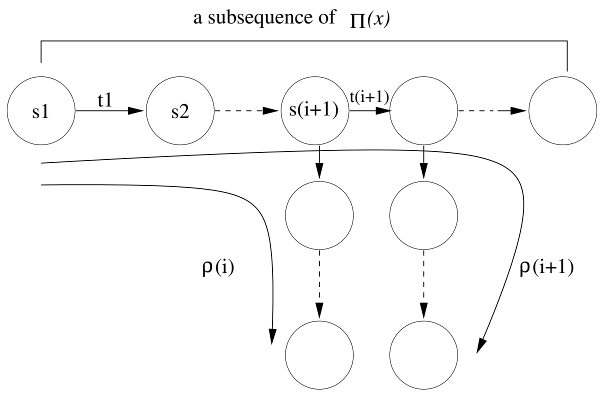


Figure 6: ρ_{i+1} is obtained from ρ_i by adding the t_{i+1} to ρ_i

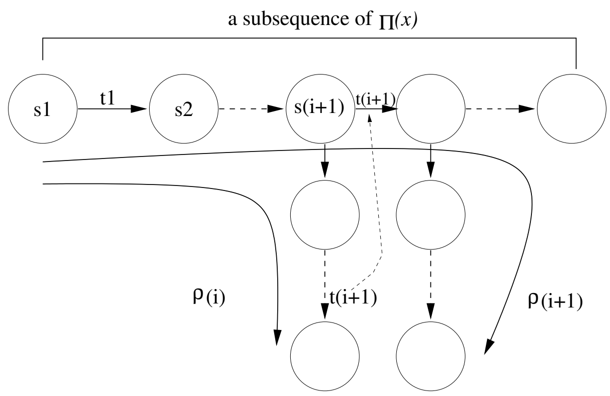


Figure 7: ρ_{i+1} is obtained from ρ_i by moving t_{i+1} into the appropriate position

At the end of the construction, the first $m-1$ transitions of ρ_{m-1} are $s_1 \xrightarrow{t_1} s_2 \ldots \xrightarrow{t_{m-1}} s_m$, t_i is not in ρ_{m-1}, and u is disabled at every state after s_m in ρ_{m-1} (observation **O3**). In other words, u is not in ρ_{m-1}. From **C2**, we can conclude that u is not in $\rho_0 = \rho$, which contradicts (**O4**). $\square$

Lemma 4 Let σ be a (finite or infinite) sequence from a state x in G_f. If x is also in V_r, then there is a sequence ρ from x in G_r that satisfies exactly the same set of LTL-X formulae on global transitions as σ.

Proof: The proof is by constructing a ρ that satisfies the lemma. This construction is very similar to the construction in Lemma 3. The construction is by transforming σ successively in σ_1, σ_2 ... such that at each step, the validity of LTL formulae are not affected, and the last sequence is ρ (if σ is infinite the construction is also infinite). If σ contains no transitions (i.e., $\sigma = x$), then ρ is equal to σ. Otherwise, let $\sigma = x \xrightarrow{a} y \ldots \sigma(\text{inf})$, i.e., let the first transition be a.
Case 1: x is either expanded by `Twophase` in phase 2 or x is expanded in phase 1 by transition a. From the algorithm it is clear that $y \in V_r$. In this case, ρ also starts with a, and the $\rho(2) \ldots \rho(\text{inf})$ is obtained by this construction from y and $\sigma(2) \ldots \sigma(\text{inf})$.
Case 2: x is expanded in phase 1 by transition b_1 different from a. Let Π_x (Lemma 2) be the finite sequence $(x = s_1) \xrightarrow{b_1} \ldots s_j \xrightarrow{b_j} s_{j+1}$.
Case 2.1: a is in $\{b_1 \ldots b_j\}$. Let t be the smallest $1 \le t \le j$ such that $b_t = a$. In this case let sequence p be $(x = s_1) \xrightarrow{b_1} \ldots s_{t-1} \xrightarrow{b_{t-1}} s_t$. (Construction continues at "Case 2 (Contd)" below.)
Case 2.2: a is not in $\{b_1 \ldots b_j\}$. In this case, let t be $j+1$, and p be Π_x (i.e., $p = (x = s_1) \xrightarrow{b_1} \ldots \xrightarrow{b_{t-1}} s_t$). (Construction continues at "Case 2 (Contd)" below.)
Case 2 (Contd): By construction, $p = (x = s_1) \xrightarrow{b_1} \ldots s_{t-1} \xrightarrow{b_{t-1}} s_t$ is in G_r, and $s_t \xrightarrow{a} a(s_t)$ is in G_r. By Lemma 3, all transitions in p are independent of a. Now $\sigma_1, \sigma_2 \ldots \sigma_{t-1}$ are constructed such that σ_i and p are identical up to the first i transitions. (Since p is in G_r, the $\sigma_i(1) \ldots \sigma_i(i+1)$ is also in G_r). Let σ_0 be σ. σ_{i+1} is obtained from σ_i as follows.
Case 2.a: b_{i+1} does not occur in $\sigma_i(i+1) \ldots \sigma_i(\text{inf})$. From Lemma 3, b_{i+1} is independent of all transitions in $\sigma_i(i+1) \ldots \sigma(\text{inf})$. σ_{i+1} obtained by inserting b_{i+1} into σ_i at position $i+1$. From Note 1, σ_i and σ_{i+1} satisfy the same set of LTL-X formulae on global transitions. (Construction continues at "Case 2 (Contd)" below.)
Case 2.b: b_{i+1} *first* appears in $\sigma_i(i+1) \ldots \sigma_i(\text{inf})$ at l th position. Again from Lemma 3, b_{i+1} is independent of all transitions in $\sigma_i(i+1) \ldots \sigma_i(l-1)$. In this case, σ_{i+1} is obtained from σ_i by moving b_{i+1} from lth position to the $i+1$th position. By Note 2, σ_i and σ_{i+1} satisfy the same set of LTL-X formulae on global transitions. (Construction continues at "Case 2 (Contd)" below.)
Case 2 (Contd): By construction, the first $t-1$ transitions of σ_{t-1} are the transitions of p, and the t th transition is a. The initial segment of ρ will be the first t transitions of σ_{t-1}, i.e., $(x = s_1) \xrightarrow{b_1} s_2 \ldots s_{t-1} \xrightarrow{b_{t-1}} s_t \xrightarrow{a} a(s_t)$. From the construction of p, it is clear that this segment is in G_r. $\rho(t+2) \ldots \rho(\text{inf})$ is obtained by recursively applying this construction to the sequence $\sigma_{t-1}(t+2) \ldots \sigma_k(\text{inf})$ from the state $a(s_{t+1})$. $\square$

Theorem 1 Let ϕ be a LTL-X formulae on global transitions. ϕ holds in G_f from the initial state iff it holds in G_r generated by `Twophase`.

Proof: If ϕ is true in G_f, then since G_r is a subgraph of G_f, it is also true in G_r. If ϕ is false in G_f, let σ be a sequence starting from initial state that shows the violation. Since initial state is added to V_r, by the above lemma, a $\rho \in G_r$ can be constructed that reveals the violation of ϕ. $\square$

5 On-the-Fly Model-checking

A model-checking algorithm is said to be on-the-fly if it examines the state graph of the system as it builds the graph to find the truth value of the property under consideration. If the truth value of the property can be evaluated by inspecting only a subgraph, then the algorithm need not generate the entire graph. Since state graph of many protocols is quite large, an on-the-fly model-checking algorithm might be able to find errors in protocols that are otherwise impossible to analyze.

As discussed in Section 2.1, the model checking problem $M \models \phi$ can be equivalently viewed as answering the question if the graph represented by $S = M \otimes A_{\neg\phi}$, the synchronous product of the model M and the Büchii automaton representing $\neg\phi$, does not contain any paths satisfying the acceptance condition of $A_{\neg\phi}$. The algorithms `dfs` and `dfs_po` are not on-the-fly model-checking algorithms since they construct the graph in E_f or E_r, which must be analyzed later to find if the acceptance condition of the Büchii automaton $A_{\neg\phi}$ is met or not. Note that E_f and E_r holds the information about the edges traversed as part of the search.

The condition that there is an infinite path in E (E_f or E_r) that satisfies the acceptance condition of $A_{\neg\phi}$ can be equivalently expressed as there is a strongly connected component (SCC) in the graph that satisfies the acceptance condition. Tarjan [Tar72] presented a DFS based on-the-fly algorithm to compute SCCs *without storing any edge information*. Since space is at a premium for most verification problems, not having to store the edge information can be a major benefit of using this algorithm. This algorithm uses one word overhead per state visited and traverses the graph twice.

[CVWY90] presents an on-the-fly model-checking algorithm, shown in Figure 8, to find if a graph has at least one infinite path satisfying a Büchii acceptance condition. Note that while Tarjan's algorithm can find all strongly connected components that satisfy the acceptance condition of $A_{\neg\phi}$, [CVWY90] algorithm is guaranteed to find only one infinite path satisfying the acceptance condition. Since presence of such an infinite path implies that the property is violated, it is usually sufficient to find one infinite path. The attractiveness of the [CVWY90] algorithm comes from the fact that it can be implemented with only one bit per state compared to one word per state in the case of Tarjan's algorithm. This algorithm, shown in Figure 8, consists of two DFS searches, `dfs1` and `dfs2`. The outer dfs, `dfs1`, is very similar to `dfs`, except that instead of maintaining E_f, the algorithm calls an inner dfs, `dfs2`, after an accept state is fully expanded, and `dfs2` finds if that accept state can reach itself by expanding the state again. If the state can reach it self, then a path

```
model_check()
{
    V1 := φ; V2 := φ;
    dfs1(InitialState);
}
```

```
/* outer dfs */
dfs1(s)
{
    V1 := V1 + {s};
    [1] foreach enabled transition t do
        if t(s) ∉ V1 then
            dfs(t(s));
        endif;
    endforeach;
    [2] if s is an accept state and
        /* Call nested dfs */
        s ∉ V2 then
            seed := s;
            dfs2(s);
        endif;
    endif;
}
```

```
/* inner dfs */
dfs2(s)
{
    V2 := V2 + {s};
    [1] foreach enabled transition t do
        if t(s)=seed then error();
        elseif t(s) ∉ V2 then
            dfs2(t(s));
        endif;
    endforeach;
}
```

Figure 8: On-the-Fly Model-checking algorithm

violating ϕ can be found from the stack needed to implement `dfs1` and `dfs2`.

This figure assumes that full state graph is being generated. To use it along with partial order reductions, statements labeled $\boxed{1}$ can be appropriately modified to use the transitions in `ample(s)` (when used in conjunction with `dfs_po`) or with the search strategy of Two phase. Earlier attempts at combining this on-the-fly model-checking algorithm with the `dfs_po` have been shown to incorrect in [HPY96]. The reason is that `ample(s)` depends on `redset`, hence when a state s is expanded on the highlighted lines in `dfs1` and `dfs2`, `ample(s)` might evaluate to different values. If `ample(s)` returns the different set of transitions in `dfs1` and `dfs2`, even if an accept state s is reachable from it self in the graph constructed by `dfs1`, `dfs2` might not be able to prove that fact. Since the information in `redset` is different for `dfs1` and `dfs2`, `ample(s)` may indeed return different transitions, leading to an incorrect implementation. [HPY96] solves the problem using the following scheme: `ample(s)` imposes an ordering on the processes in the system. When `ample(s)` cannot choose a process, say P_i, in `dfs1` due to the proviso, they choose `ample(s)` to be equal to all enabled transitions of s. In addition, one bit of information is recorded in `V1` to indicate that s is completely expanded. When `s` is encountered as part of `dfs2`, this bit is inspected to find if `ample(s)` must return all enabled transitions or if it must return a subset of transitions *without requiring the proviso*. This strategy reduces the opportunities for obtaining effective reductions, but it is deemed a good price to pay for the ability to use the on-the-fly model-checking algorithm.

Thanks to the independence of `phase1` on global variables including V_r, when `phase1(s)` is called in dfs2, the resulting state is exactly same as when it is called in dfs1. Hence the on-the-fly model-checking algorithm can be used easily in conjunction with Two phase. In Section 5.2, it is argued that the combination of this on-the-fly model-checking algorithm, the selective caching technique can be used *directly* with Two phase.

5.1 Selective caching

Both Twophase and dfs_po, when used in conjunction with the [CVWY90] algorithm, obviate the need to maintain E_r. However, memory requirements to hold V_r, for most practical protocols, can be still quite high. Selective caching refers to the class of techniques where instead of saving every state visited in V_r, only a subset of states are saved.

There is a very natural way to incorporate a selective caching into Twophase. Instead of adding list to V_r only s can be added. This guarantees that a given state always generates the same subgraph beneath it whether it is expanded as part of outer dfs or inner dfs, hence the [CVWY90] can still be used. Adding s instead of list also means that the memory used for list in phase1 can be reused. Even the memory required to hold the intermediate variable list can be reduced: the reason for maintaining this variable is only to ensure that the while loop terminates. This can be still guaranteed if instead of adding s to list unconditionally, it is added only if "s<olds", where $<$ is any total ordering on $\mathcal{S}$. PV uses bit-wise comparison as $<$.

5.2 Combining on-the-fly model-checking and selective caching with Two phase

When the selective caching technique is combined with Two phase, the execution goes as follows: a given state is first expanded by phase1, then the resulting state is added to V_r and fully expanded. In other words, V_r contains only fully expanded states, which implies that the state graph starting a given state is the same in dfs1 and dfs2 of the on-the-fly algorithm. Hence, the on-the-fly algorithm and selective caching can be used together with Two phase.

6 Experimental Results

As already mentioned, Twophase outperforms the proviso based algorithm dfs_po (implemented in SPIN) when the proviso is invoked often, confirmed by the results in Table 1. This table shows results of running dfs_po and Twophase (with and without selective caching enabled) on various protocols. The column corresponding to dfs_po shows the number of states entered in V_r and the time taken in seconds by the SPIN. The column "all" column in Twophase shows the number of states in V_r and the time taken in seconds when Twophase is run *without* the selective caching.

Protocol	dfs_po	Twophase	
		all	Selective
B5	243/0.34	11/0.33	1/0.3
W5	63/0.33	243/0.39	243/0.3
SC3	17,741/4.6	2,687/1.6	733/1.4
SC4	749,094/127	102,345/41.0	47,405/21.9
Mig	113,628/14	22,805/2.6	9,185/1.7
Inv	961,089/37	60,736/5.2	27,600/3.0
Pftp	95,241/11.0	187,614/30	70,653/19
Snoopy	16,279/4.4	14,305/2.7	8,611/2.4
WA	4.8e+06/340	706,192/31	169,680/21
UPO	4.9e+06/210	733,546/32	176,618/21
ROWO	5.2e+06/330	868,665/44	222,636/32

Table 1: Number of states visited and the time taken in seconds by the `dfs_po` algorithm and `Twophase` algorithm on various protocols

The "Selective" column in `Twophase` shows the number of states entered in V_r *or* `list` and time taken in seconds when `Twophase` is run with the selective caching. All verification runs are conducted on an Ultra-SPARC-1 with 512MB of DRAM.

Contrived examples: B5 is the system shown in Figure 4(a) with 5 processes. W5 is a contrived example to show that `Twophase` does not always outperform the `dfs_po`. This system has no deterministic states; hence `Twophase` degenerates to a full search, while `dfs_po` can find significant reductions. *SC* is a server/client protocol. This protocol consists of n servers and n clients. A client chooses a server and requests for a service. A service consists of a two round trip messages between server and client and some local computations. `dfs_po` cannot complete the graph construction for $n = 4$, when the memory is limited to 64MB; when the memory limit is increased to 128MB it generates 750k states.

DSM protocols: *Mig* and *inv* are two cache coherency protocols used in the implementation of distributed shared memory (DSM) using a directory based scheme in Avalanche multiprocessor [CKK96]. In a directory based DSM implementation, each cache line has a designated node that acts as its "home", i.e., the node that is responsible for maintaining the coherency of the line. When a node needs to access the line, if it does not have the required permissions, it contacts the home node to obtain the permissions. Both *mig* and *inv* have two cache lines and four processes; two processors act as home nodes for the cache lines and the other two processors access the cache lines. Both algorithms can complete the analysis of *Mig* within 64MB of memory, but on *inv*, `dfs_po` requires 128MB of memory `Twophase` on the other hand finishes comfortably generating a modest 27,600 states (with selective caching) or 60,736 states (without selective caching) in 64MB.

Protocols in SPIN distribution: *Pftp* and *snoopy* protocols are provided as part of SPIN distribution. On *pftp*, `dfs_po` generates fewer states than `Twophase` without state caching. The reason is that there is very little determinism in this protocol. Since `Twophase` depends on determinism to bring reductions, it generates a larger state space. However, with state caching, the number of states in the hash table goes down by a factor of 2.7. On *snoopy*, even though `Twophase` generates fewer states, the number of states generated `dfs_po` and `Twophase` (without selective caching) is very close to obtain any meaningful conclusion. The reason for this is two-fold. First, this protocol contains some determinism, which helps `Twophase`. However, there are a number of deadlocks in this protocol. Hence, the proviso is not invoked many times. Hence the number of states generated is very close.

Memory model verification examples: *WA*, *UPO*, and *ROWO* test the interaction of PA (Precision Architecture from Hewlett-Packard) memory ordering rules with the runway bus protocol [BCS96, GGH+97]. Runway is a high-performance split-transaction bus designed to support cache coherency protocols required to implement a symmetric multiprocessor (SMP). These three protocols consist of two HP PA models connected to the runway bus, executing read and write instructions. These property of interest is whether the PA/runway system correctly implements memory consistency rules called write atomicity (WA), and uniprocessor ordering (UPO), and read-order, write-order (ROWO) [Col92]. On these protocols, the number of states saved by `dfs_po` is approximately 25 times larger than the number of states saved by `Twophase` (with selective caching).

7 Conclusion

We presented a new partial order reduction algorithm Two phase that does not use the proviso, and formally proved that it preserves all LTL-X properties on global variables. We also showed how the algorithm can be combined with an on-the-fly model-checking algorithm. Since the algorithm does not use the proviso, it outperforms previous algorithms on protocols where the proviso is invoked often. Two phase also naturally lends itself to be used in conjunction with a simple yet powerful selective caching scheme. Two phase is implemented in a model-checker called PV which can be obtained by contacting the authors.

References

[ABHQ97] R. Alur, R. K. Brayton, T. A. Henzinger, and S. Qadeer. Partial-order reduction in symbolic state space exploration. *Lecture Notes in Computer Science*, 1254, 1997.

[BCS96] William R. Bryg, Kenneth K. Chan, and Nicholas S.Fiduccia. A high-performance, low-cost multiprocessor bus for workstations and midrange servers. *Hewlett-Packard Journal*, pages 18–24, February 1996.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[CKK96] John B. Carter, Chen-Chi Kuo, and Ravindra Kuramkote. A comparison of software and hardware synchronization mechanisms for distributed shared memory multiprocessors. Technical Report UUCS-96-011, University of Utah, Salt Lake City, UT, USA, September 1996.

[Col92] W. W. Collier. *Reasoning About Parallel Architectures*. Prentice-Hall, Englewood Cliffs, NJ, 1992.

[CVWY90] C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis. Memory efficient algorithms for the verification of temporal properties. In *Computer Aided Verification*, pages 233–242, June 1990.

[Dil96] David Dill. The stanford murphi verifier. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 390–393, New Brunswick, New Jersey, July 1996. Springer-Verlag. Tool demo.

[DPN93] David L. Dill, Seungjoon Park, and Andreas Nowatzyk. Formal specification of abstract memory models. In Gaetano Borriello and Carl Ebeling, editors, *Research on Integrated Systems*, pages 38–52. MIT Press, 1993.

[GGH+97] G. Gopalakrishnan, R. Ghughal, R. Hosabettu, A. Mokkedem, and R. Nalumasu. Formal modeling and validation applied to a commercial coherent bus: A case study. In Hon F. Li and David K. Probst, editors, *CHARME*, Montreal, Canada, 1997.

[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.

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

[GW92] P. Godefroid and P. Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. In Kim G. Larsen and Arne Skou, editors, *Computer Aided Verification*, volume 575 of *LNCS*, pages 332–342, Berlin, Germany, July 1992. Springer.

[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.

[Hol97] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997. Special issue on Formal Methods in Software Practice.

[HP94] Gerard Holzmann and Doron Peled. An improvement in formal verification. In *Proceedings of Formal Description Techniques*, Bern, Switzerland, October 1994.

[HP96] Gerard J. Holzmann and Doron Peled. The state of SPIN. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 385–389, New Brunswick, New Jersey, July 1996. Springer-Verlag. Tool demo.

[HPY96] G.J. Holzmann, D. Peled, and M. Yannakakis. On nested depth first search. In *The* SPIN *Verification System*, pages 23–32. American Mathematical Society, 1996. Proc. of the Second SPIN Workshop.

[KLM+97] Robert P. Kurshan, Vladimir Levin, Marius Minea, Doron Peled, and Husnu Yenigun. Verifying hardware in its software context. In *International Conference on Computer Aided Design*, San Jose, CA, USA, 1997.

[Kur94] R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*. Princeton University Press, 1994.

[Lip75] Richard J. Lipton. Reduction: A method of proving properties of parallel programs. *CACM*, 18(12):717–721, December 1975.

[NG96] Ratan Nalumasu and Ganesh Gopalakrishnan. Partial order reductions without the proviso. Technical Report UUCS-96-008, University of Utah, Salt Lake City, UT, USA, August 1996.

[NG97a] Ratan Nalumasu and Ganesh Gopalakrishnan. A new partial order reduction algorithm for concurrent system verification. In *CHDL*, pages 305 – 314, Toledo, Spain, April 1997. Chapman Hall, ISBN 0 412 78810 1.

[NG97b] Ratan Nalumasu and Ganesh Gopalakrishnan. PV: a model-checker for verifying ltl-x properties. In *Fourth NASA Langley Formal Methods Workshop*, pages 153–161. NASA Conference Publication 3356, 1997.

[NK95] Ratan Nalumasu and Robert P. Kurshan. Translation between S/R and Promela. Technical Report ITD-95-27619V, Bell Labs, July 1995.

[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.

[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. also in Computer Aided Verification, 1994.

[Tar72] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, June 1972.

[Val92] Antti Valmari. A stubborn attack on state explosion. *Journal of Formal Methods in Systems Design*, 1:297–322, 1992. Also in Computer Aided Verification, 1990.

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

[vL90] J. van Leeuwen, editor. *Handbook of Theoretical Computer Science, Volume B:Formal Models and Semantics*. Elsevier / MIT Press, 1990.