

An Efficient Partial Order Reduction Algorithm with an Alternative Proviso Implementation

Ratan Nalumasu[‡] and Ganesh Gopalakrishnan[§]

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

Abstract. This paper presents a partial order reduction algorithm called **Twophase** that generates a significantly reduced state space on a large class of practical protocols over alternative algorithms in its class. The reduced state-space generated by **Twophase** preserves all CTL*-X assertions. **Twophase** achieves this reduction by following an alternative implementation of the *proviso* step. In particular, **Twophase** avoids the *in-stack* check that other tools use in order to realize the *proviso* step. In this paper, we demonstrate that the *in-stack* check is inefficient in practice, and demonstrate a much simpler alternative method of realizing the proviso. **Twophase** can be easily combined with an on-the-fly model-checking algorithm to reduce memory requirements further. A simple but powerful selective caching scheme can also be easily added to **Twophase**.

A version of **Twophase** using on-the-fly model-checking and selective caching has been implemented in a model-checker called PV (Protocol Verifier) and is in routine use on large problems. PV accepts a proper subset of Promela and a **never** automaton expressing the LTL-X assertion to be verified. PV has helped us complete full state-space search several orders of magnitude faster than all alternative tools available in its class on dozens of real protocols. PV has helped us detect bugs in real Distributed Shared Memory cache coherency protocols that were missed during incomplete search using alternate tools.

0.0.0.1. *Keywords:* Partial order reductions, explicit enumeration, temporal logic, on-the-fly model-checking, proviso, in-stack 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,

[‡] Supported in part by a Graduate Fellowship from the University of Utah. Present address: Hewlett-Packard, Cupertino Systems Lab, Cupertino, CA, 95014 - ratan@cup.hp.com

[§] Supported in part by DARPA under contract #DABT6396C0094 (Utah Verifier), and NSF grant CCR-9800928



[Hol91, Dil96, HP96, Hol97] has shown considerable promise in real-world protocol verification problems and has 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 often increases 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, whereas 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

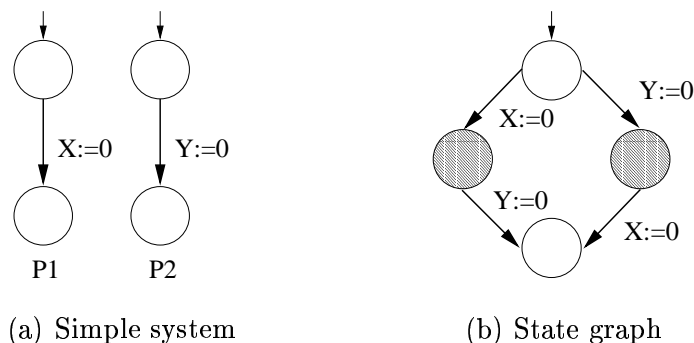


Figure 1. A simple system and its state graph

factor. This paper presents a new partial order reduction algorithm called *Twophase*, 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.

The partial order reduction algorithm presented in [Pel96a, HP94] is implemented in the explicit state enumeration model checker SPIN¹ [HP96, Hol97] and in the implicit state exploration tools VIS and COSPAN [ABHQ97, KLM⁺97, NK95]. The algorithm presented in [God95] is implemented in the PO-PACKAGE tool. In general, partial order reduction algorithms must avoid the so called *ignoring problem*—the problem of infinitely postponing an action, resulting in an incorrect reduction—using some measure of progress. Partial order reduction algorithms generally solve the problem by using *provisos*, whose need was first recognized by Valmari [Val92]. The traditional way of implementing provisos (made clear momentarily) is to ensure 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 PO-PACKAGE algorithm (and also the algorithm presented in [HGP92]) requires that at least one of the selected transitions do not generate a state in the stack, whereas SPIN algorithm requires the stronger condition that no selected transition generates a state in the stack. The stronger proviso (hereafter abbreviated as proviso) is sufficient to preserve all stutter free linear time temporal logic (LTL-X) formulae (safety and liveness), whereas the weaker proviso preserves only stutter free safety properties [HGP92, HP94, Pel93, Pel96a].

In the context of verifying reactive systems such as directory based coherence protocols and server-client protocols, we observed that the most commonly adopted method of implementing the proviso—namely an explicit in-stack check of the next state(s)—causes all existing partial order reduction algorithms to be ineffective [NG96]. As an example, on *invalidate*, a distributed shared memory protocol described later, SPIN aborts its search by running out of memory after generating more than 270,000 states when limited to 64MB memory usage, ultimately finishing in 961,089 states when allowed more memory. PO-PACKAGE also aborts its search after generating a similar number of states. In *invalidate*, there are many opportunities for partial order reductions to

¹ All references to SPIN in this paper refer to versions that existed in the public domain between the end of 1995, the first time *Twophase* was implemented within PV, and mid-1998, when most of the experiments comparing PV and SPIN were finished.

reduce the complexity; hence, protocols of this complexity ought to be easy for on-the-fly explicit enumeration tools to handle. This intuition is confirmed by the fact that **Twophase**, a partial order reduction that does not use the in-stack check, finishes comfortably on this protocol within 64M main memory, generating only 27,600 states. In fact, as shown in Section 7, in all nontrivial examples, our implementation of **Twophase** in our model checker called PV [NG98b] outperforms in-stack checking based algorithms.

Alternate Implementation of Provisos

The term *proviso* is used to refer to condition A5 of [Val96, Page 222]², which, roughly speaking, states that every action enabled in a state s of the reduced state space is present in the stubborn set of a state s' of the reduced state-space reachable from s . The **Twophase** algorithm implements the proviso condition A5 as follows. When it encounters a new state x , it expands the state using only *deterministic* transitions in its *first phase* (both these notions will be defined shortly), resulting in a state y . Deterministic transitions, equivalent to singleton ample sets [Pel96a], are those that can be taken at the state without effecting the truth of the property being verified. Then in the *second phase*, y is expanded *completely*. The need to cross-over from the first-phase to the second phase can be detected using a different (and much simpler) strategy than an in-stack check. This strategy provides **Twophase** all its advantages mentioned earlier.

An important consequence of the above strategy is that **Twophase** 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 it 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 due to the need to share information among these three aspects. In [HPY96], it is shown that previous attempts at combining in-stack checking based algorithms with the on-the-fly algorithm presented in [CVWY90] have been erroneous. However, thanks to the fact that the first phase of **Twophase** does not depend on the stack state, it can be combined easily with the

² When we use the colloquial or the plural form of the word “proviso,” we will mean particular implementations, such as in-stack checks.

on-the-fly algorithm presented in [CVWY90] and a simple but effective selective-caching strategy, as discussed in Sections 6 and 6.1.

The rest of the paper is organized as follows. Section 2 presents related work. Section 3 presents definitions and background. Section 4 presents the basic depth-first search algorithm and the in-stack checking based partial order reduction algorithm. Section 5 presents the **Twophase** algorithm and its proof of correctness. Section 6 presents the on-the-fly model checking algorithm presented in [CVWY90] and discusses on how it can be combined with **Twophase**. This section also presents a very simple but effective selective caching strategy and shows how it can be incorporated into **Twophase**. Section 7 compares the performance of the algorithm [Pel96a] implemented in SPIN with that of **Twophase** implemented in the PV tool and provides a qualitative explanation of the results. Finally, Section 8 provides concluding remarks.

2. Related Work

Lipton [Lip75] suggested a technique to avoid exploring the entire state graph to find if a concurrent system deadlocks. Lipton noted 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.

In [GW92, GP93, God95], a partial order theory based on traces to preserve safety properties is presented. This work uses a slight variation of the proviso. In [Pel96a], a partial order reduction algorithm based on *ample* sets and the proviso is presented. In [HP94], an algorithm very similar to (and based on the algorithm of [Pel96a]) is given. This algorithm is implemented in SPIN. The algorithm in [Pel96a] is discussed in Section 4. In all these algorithms, the proviso is realized using an in-stack check. 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 **Twophase** algorithm was conceived at the end of 1995 in the context of verifying real distributed shared memory protocols used in the Avalanche processor [CKK96]. We first proved that **Twophase** preserved stutter-free safety properties [NG97a], and later extended the proof to LTL-X [NG97b, NG98a, Nal98]. The PV tool embodying **Twophase** was also demonstrated [NG98b].

Thanks to editorial comments received during the review of this paper, we now know that the condition ‘deterministic’ discovered by us to avoid the explicit in-stack check turns out to be the exact same condition required by CTL*-X (stutter free CTL*) preserving methods [GKPP95, Val96, Pel96b]. In fact, we now realize that `Twophase` ends up implementing conditions A5 and A8 of [Val96]. Thus, `Twophase` preserves CTL*-X, as argued in Section 5.1 referring to existing proofs. With these new insights, we can characterize our contributions as follows.

1. An experimental characterization of the state explosion caused by the in-stack method.
2. A new partial order reduction algorithm called `Twophase` that uses an alternative proviso implementation to the in-stack method, thereby considerably mitigating state explosion.
3. Advantages of our method of realizing the proviso in terms of supporting selective caching and on-the-fly model-checking.
4. An extensive list of experiments that demonstrate the superior performance of the PV model-checker compared to other tools in its class.

3. Definitions and Notation

A process oriented modeling language with each process maintaining a set of local variables that only it can access is assumed. The value of these local variables form the *local state* of the process. For convenience, each process is assumed to contain a distinguished local variable called program counter (“control state”). A concurrent system (“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 (“state”) 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 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 nonempty). If a transition t is enabled in a state $s \in \mathcal{S}$, then it is uniquely defined. Nondeterminism can be simulated by having multiple transitions from a given program counter. t, t' are used to indicate transitions, $s \in \mathcal{S}$ 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 $\text{pc}(s, P)$ to indicate the program counter (control state) of P in s , and $\text{pc}(t)$ 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, whereas 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 a 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, Pel96a, HP94, God95] use the notion of *ample set* based on *safe* transitions. **Twophase**, on the other hand, uses the notion of *deterministic*—singleton ample sets—to obtain reductions. The proof of correctness of the **Twophase** algorithm uses the notion of *independent* transitions.

3.1. 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:

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

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 nondeterministic 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].

4. Basic DFS and In-stack check Based Partial Order Reduction Algorithms

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

Partial order reduction based search algorithms attempt to replace 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 algorithm in [Pel96a, HP94] is shown as `dfs_po` Figure 3. As already mentioned, this algorithm is implemented in SPIN. 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

```

model_check()                dfs(s)
{                               {
    Vf := ϕ;                   Vf := Vf + {s};
    Ef := ϕ;                   [1] foreach enabled t in s do
    dfs(InitialState);         [2] Ef := Ef + {(s,t,t(s))};
}                               if t(s) ∉ Vf then
                                dfs(t(s));
                                endif
                                endforeach
}

```

Figure 2. Basic depth first search algorithm

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 without this condition the algorithm may never explore 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 $t(s)$ is itself being explored (*i.e.* $t(s) \in \text{redset}$), 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 Section 6.1 shows later, 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.

4.1. 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, Pel96a]. However, in many practical protocols, the reductions are not as effective as they can be. The reason can be traced to the implementation of the proviso using in-stack checking. 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.* $\tau_1 \dots \tau_4$ commute with $\tau_5 \dots \tau_8$. 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 in Figure 3. This graph is obtained as follows. The initial state is $\langle s_0, s_0 \rangle$. `ample($\langle s_0, s_0 \rangle$)` may return either

```

dfs_po(s)                                ample(s)
{
  /* Record in redset that
     s is partly expanded */
  redset := redset + {s};
  Vr := Vr + {s};
  /* ample(s) uses redset */
  [1] foreach transition t
      in ample(s) do
    [2] Er := Er + {s, t, t(s)};
    if t(s) ∉ Vr then
      dfs_po(t(s));
    endif;
  endforeach;
  /* s is completely expanded.
     So remove from redset */
  redset := redset - {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

$\{\tau_1, \tau_3\}$ or $\{\tau_5, \tau_7\}$. Without loss of generality, assume that it returns $\{\tau_1, \tau_3\}$, resulting in states $\langle s1, s0 \rangle$ and $\langle s2, s0 \rangle$. Again, without loss of generality, assume that the algorithm chooses to expand $\langle s1, s0 \rangle$ first, where transitions $\{\tau_2\}$ of P_1 and $\{\tau_5, \tau_7\}$ of P_2 are enabled. $\tau_2(\langle s1, s0 \rangle) = \langle s0, s0 \rangle$, and when $\text{dfs_po}(\langle s1, s0 \rangle)$ is called, $\text{redset} = \{\langle s0, s0 \rangle\}$. As a result $\text{ample}(\langle s1, s0 \rangle)$ cannot return $\{\tau_2\}$; it returns $\{\tau_5, \tau_7\}$. Executing τ_5 from $(\langle s1, s0 \rangle)$ results in $\langle s1, s1 \rangle$, 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 an in-stack checking based partial order reduction algorithm might fail to bring appreciable reductions. As confirmed by the examples in Section 7, the algorithm may not bring much reductions in realistic protocols also.

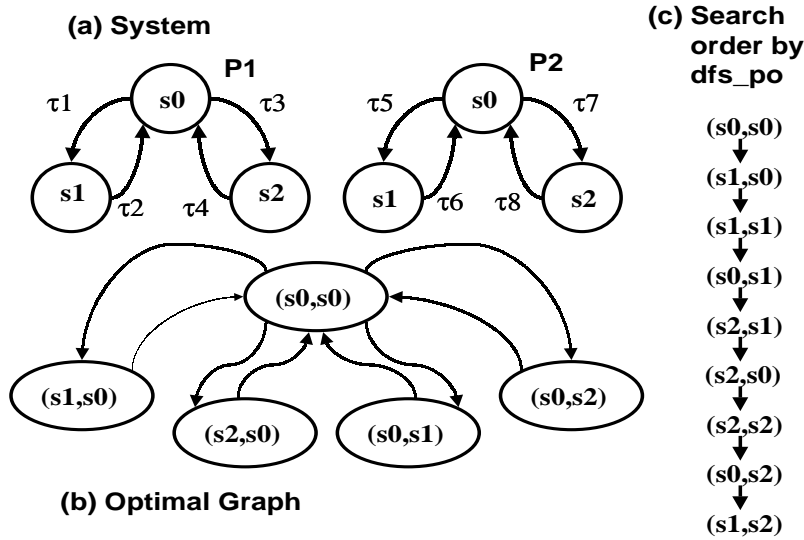


Figure 4. A simple example, its optimal reduced graph, and the reduced graph generated by dfs_po

5. The Twophase Algorithm

As the previous contrived example shows, the size of the reduced graph generated by an algorithm based on in-stack checking 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 in-stack checking, 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 7 will demonstrate that in realistic systems also the number of extra states generated due to the proviso can be high.

The proposed algorithm is described in Figure 5. 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. The **Twophase** algorithm often outperforms SPIN (and PO-PACKAGE) as evidenced by the examples in Section 7. Note that **phase1** is *more gen-*

```

model_check()
{
   $V_r := \phi$ ;
   $E_r := \phi$ ;
  /* fe (fully expanded) is used in proof */
  fe :=  $\phi$ ;
  Twophase(initial_state);
}

phase1(in)                                Twophase(s)
{                                          {
  s := in;                                /* Phase 1 */
  list := {s};                             (path, s) := phase1(s);
  path := {};
  foreach process P do                    /* Phase 2: Classic DFS */
    while (deterministic(s, P))          if  $s \notin V_r$  then
      /* Let t be the only enabled      /* fe is used in proof */
      transition in P */                [1]  $V_r := V_r + \text{all states in path} + \{s\}$ ;
      olds := s;                        [2]  $E_r := E_r + \text{path}$ ;
      s := t(olds);                      [3] fe := fe + {s};
      path := path +                    foreach enabled transition t do
      {(olds, t, s)};                    [3]  $E_r := E_r + (s, t, t(s))$ ;
      if (s  $\in$  list)                    if t(s)  $\notin V_r$  then
        goto NEXT_PROC;                  Twophase(t(s));
      endif                                endif;
      [1] list := list + {s};            endforeach;
      endwhile;                          else
      NEXT_PROC:                          [1']  $V_r := V_r + \text{all states in path}$ ;
      /* next process */                  [2']  $E_r := E_r + \text{path}$ ;
    endforeach;                          endif;
  return(path, s);                        }
}

```

Figure 5. The Twophase algorithm

eral than the notion of *coarsening* actions (for example, implemented as a `d_step` in SPIN). In coarsening, 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.

5.1. CORRECTNESS OF THE TWOPHASE ALGORITHM

The correctness of **Twophase** follows from previous results. In particular, Theorem 6.3 of [Val96] states that if conditions $\ddot{A}5$ and $\ddot{A}8$ hold, the reduced and the unreduced transition systems are branching-bisimilar. Here, condition $\ddot{A}5$ states that every action enabled in a state s of the reduced state space is eventually in the stubborn set of a state s' of the reduced state-space that is reachable from s . This condition is easily satisfied by **Twophase**: those states attained at the end of **phase1** are fully expanded in **phase2** (**fe** in Figure 5 under **Twophase** records all those states that are fully expanded). Condition $\ddot{A}8$ states that for every state s in the reduced state space, either its stubborn set contains all actions or there is an internal action a such that the stubborn set of s has exactly a enabled in s and further a is super-deterministic in s . The exact definition of super-determinism in the context of [Val96] may be found in that reference; in our context, super-determinism is what we defined as *deterministic* on Page 8.

The correctness of **Twophase** can be understood also in terms of the proof in [Pel96b]. A proof of correctness of **Twophase** from first principles may be found in [Nal98].

6. 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 the state graphs of many protocols are 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 3.1, 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.

The on-the-fly model checking algorithm of [CVWY90] is shown in Figure 6. This algorithm can be used to find if a graph has at least one infinite path satisfying a Büchi acceptance condition. Note that whereas Tarjan's algorithm can find all strongly connected components that satisfy the acceptance condition of $A_{\neg\phi}$, the algorithm in [CVWY90] 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 algorithm in [CVWY90] 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. The algorithm 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. `dfs2` finds if that accept state can reach itself by expanding the state again. If the state can reach it self, then a path 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 `1` in `dfs1(s)` and `dfs2(s)` can be appropriately modified to use the transitions in `ample(s)` (when used in conjunction with `dfs_po`) or with the search strategy of `Twophase`. Earlier attempts at combining this on-the-fly model checking algorithm with the `dfs_po` have been shown to be incorrect in [HPY96]. The reason is that `ample(s)` depends on `redset`; hence when a state s is expanded on lines indicated by `1` in `dfs1` and `dfs2`, `ample(s)` might evaluate to different values. If `ample(s)` returns a different set of transitions in `dfs1` and `dfs2`, even if an accept state s is reachable from itself 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 en-

```

model_check()
{
  V1 :=  $\phi$ ; V2 :=  $\phi$ ;
  dfs1(InitialState);
}

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

```

Figure 6. An on-the-fly model checking algorithm

countered 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 `Twophase`. In Section 6.2, it is argued that the combination of this on-the-fly model checking algorithm, the selective caching technique can be used *directly* with `Twophase`.

6.1. SELECTIVE CACHING

Both `Twophase` and `dfs_po`, when used in conjunction with the above on-the-fly model checking 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 selective caching into **Twophase**. Instead of adding all states of **path** to V_r (line 1 in **Twophase**) 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 above on-the-fly model checking algorithm 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 <.

6.2. COMBINING ON-THE-FLY MODEL CHECKING AND SELECTIVE CACHING WITH **TWOPHASE**

When the selective caching technique is combined with **Twophase**, 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 **Twophase**.

7. Experimental Results

As already mentioned, **Twophase** outperforms the algorithm **dfs_po** (implemented in SPIN) when in-stack checking succeeds often, as confirmed by the results in Table I³. 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 SPIN⁴.

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. The “Selective” column in **Twophase** shows the

³ All these examples as well as the PV tool distribution are available in the web URL www.cs.utah.edu/formal_verification.

⁴ These experiments were run versions of SPIN available during 1997-98.

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, whereas `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 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*—a 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 main memory, albeit with `Twophase` performing much better. On *inv*, a much more involved coherence protocol, `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) within 64MB of main memory.

Protocols in the 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 by `dfs_po` and `Twophase` (without selective caching) are too close to obtain any meaningful conclusions. The reason for this is twofold. 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.

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

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

Memory model verification examples: We modeled the Hewlett-Packard Precision Architecture (HPPA) split-transaction coherent bus called Runway [BCS96, GGH⁺97], a modern symmetric multiprocessor interconnect bus, in the common subset of Promela supported by the PV tool and SPIN. Then, we applied our approach to verify memory orderings via finite-state reachability analysis (described in [Nal98, Col92, GMNG98, NGMG98]) on this Runway bus model. Our method to establish a given memory ordering such as write atomicity (WA), uniprocessor order (UPO), read-order (RO), or read and write order (ROWO) involves writing *highly non-deterministic* ‘test automata’ to drive the bus, and verifying that the execution never causes the test automata to enter one of their error states. 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).

8. Conclusion

We presented a new partial order reduction algorithm `Twophase` that implements the proviso without using in-stack checking. The correctness of `Twophase` was shown to follow from earlier results in CTL*-X

preserving partial-order reduction methods. We also showed how the algorithm can be combined with an on-the-fly model-checking algorithm. Through an extensive set of experiments, we demonstrated that **Twophase** outperforms those algorithms that realize the proviso using in-stack checking, where the in-stack check succeeds often. **Twophase** also naturally lends itself to be used in conjunction with a simple yet powerful selective caching scheme. **Twophase** is implemented in a model-checker called PV, and is available upon request.

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 Nowatzky. 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.
- GKPP95. R. Gerth, R. Kuiper, W. Penczek, and D. Peled. A partial order approach to branching time logic model checking. In *ISTCS'95, 3rd Israel Symposium on Theory of Computing and Systems*, pages 130–139, Tel Aviv, Israel, 1995. IEEE Press.

- GMNG98. Rajnish Ghughal, Abdel Mokkedem, Ratan Nalumasu, and Ganesh Gopalakrishnan. Using "test model-checking" to verify the runway-pa8000 memory model. In *Tenth Annual ACM Symposium On Parallel Algorithms and Architectures*, pages 231–239, Puerto Vallarta, Mexico, June 1998. ACM Press. Program Chair: Phillip B. Gibbons.
- 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.
- Nal98. Ratan Nalumasu. *Formal design and verification methods for shared memory systems*. PhD thesis, University of Utah, Salt Lake City, UT, USA, December 1998.
- NG96. Ratan Nalumasu and Ganesh Gopalakrishnan. Partial order reduction without the proviso. Technical Report UUCS-96-008, Department of Computer Science, University of Utah, August 1996. Available online through NCSTRL.
- 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.
- NG98a. Ratan Nalumasu and Ganesh Gopalakrishnan. A partial order reduction algorithm without the proviso. Technical Report UUCS-98-017, University of Utah, Salt Lake City, UT, USA, August 1998.
- NG98b. Ratan Nalumasu and Ganesh Gopalakrishnan. PV: An explicit enumeration model-checker. In Ganesh Gopalakrishnan and Phillip Windley, editors, *Formal Methods in Computer Aided Design*, volume 1522 of *Lecture Notes in Computer Science*, pages 522–528, Palo Alto, CA, USA, 1998. Springer-Verlag.
- NGMG98. Ratan Nalumasu, Rajnish Ghughal, Abdel Mokkedem, and Ganesh Gopalakrishnan. The ‘test model-checking’ approach to the verification of formal memory models of multiprocessors. In Alan J. Hu and Moshe Y. Vardi, editors, *Computer Aided Verification*, volume 1427 of *Lecture Notes in Computer Science*, pages 464–476, Vancouver, BC, Canada, June 1998. Springer-Verlag.
- 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.
- Pel96a. 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.
- Pel96b. Doron Peled. Partial order reduction: Linear and branching temporal logics and process algebras. In *Partial Order Methods in Verification; DIMACS Workshop*, volume 29, pages 233–258. American Mathematical Society, July 1996. Series in Discrete Mathematics and Theoretical Computer Science.
- 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.
- Val96. Antti Valmari. Stubborn set methods for process algebras. In *Partial Order Methods in Verification; DIMACS Workshop*, volume 29, pages 213–232. American Mathematical Society, July 1996. Series in Discrete Mathematics and Theoretical Computer Science.
- vL90. J. van Leeuwen, editor. *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*. Elsevier / MIT Press, 1990.