# Performance Studies of PV: an On-the-fly Model-checker for LTL-X Featuring Selective Caching and Partial Order Reduction

***Ganesh Gopalakrishnan, Ratan Nalumasu, Robert Palmer, Prosenjit Chatterjee, and Ben Prather***

UUCS-01-004

School of Computing
University of Utah
Salt Lake City, UT 84112 USA

6 April 2001

## *Abstract*

We present an enumerative model-checker PV that uses a new partial order reduction algorithm called Twophase. This algorithm does not use the in-stack check to implement the proviso, making the combination of Twophase with on-the-fly LTL-X model-checking based on nested depth-first search, as well as with selective state caching very straightforward. We present a thorough evaluation of PV in terms of several criteria including states, memory, search depth, and runtimes. Our very encouraging results, often orders of magnitude better, are objectively explained in this paper. We also explain the different selective state caching methods supported by PV as well as its user interface geared towards verifying cache coherence protocols for conformance against formal memory models. We offer the source code of PV as well as our examples through our webpage.

**Performance Studies of PV: an On-the-fly Model-checker for LTL-X Featuring Selective Caching and Partial Order Reduction**

Ganesh Gopalakrishnan, Ratan Nalumasu,
Robert Palmer, Prosenjit Chatterjee, and Ben Prather*
{ratan@cup.hp.com},{ganesh,rpalmer,prosen,prather}@cs.utah.edu,
http://www.cs.utah.edu/formal_verification/

University of Utah, School of Computing

**Abstract.** We present an enumerative model-checker PV that uses a new partial order reduction algorithm called `Twophase`. This algorithm does not use the *in-stack* check to implement the proviso, making the combination of `Twophase` with on-the-fly LTL-X model-checking based on nested depth-first search, as well as with selective state caching very straightforward. We present a thorough evaluation of PV in terms of several criteria including states, memory, search depth, and runtimes. Our very encouraging results, often orders of magnitude better, are objectively explained in this paper. We also explain the different selective state caching methods supported by PV, as well as its user interface geared towards verifying cache coherence protocols for conformance against formal memory models. We offer the source code of PV as well as our examples through our webpage.

## 1 Introduction

PV is an enumerative model-checker developed by our group. While similar to SPIN[1] in many ways, it also differs in many respects, the important one being the use of a different partial order reduction algorithm in PV called `Twophase` [Nal98,NG01]. `Twophase` implements the *ample set* calculation and the *proviso* condition (that prevents the *ignoring problem*) in a different, and much simpler way. In particular, `Twophase` does not use the *in-stack check* method to implement the proviso. This paper presents the traditional in-stack checking based algorithm and the `Twophase` algorithm, and compares them along three axes: their overall behavior, the ease of combining with nested DFS based model-checking and selective state caching, and empirical performance. Overall features of the PV tool are also discussed. The results reported in this paper are briefly discussed in the following paragraphs.

From our presentation of the two kinds of partial order reduction algorithms, it will be evident that algorithms based on in-stack checking have a tendency to leave processes hanging in their penultimate states, instead of fully resetting them to the "top level." This tendency can contribute to state explosion. We illustrate this phenomenon on both contrived as well as realistic examples.

* Supported by NSF Grants CCR-9987516 and CCR-0081406, and a gift from the Intel Corporation

[1] In this paper, "SPIN" refers to Version 3.4.3.

It is known that combining partial order reduction algorithms that use in-stack checking with nested depth-first search used to realize on-the-fly LTL-X model-checking [CVWY90] is quite tricky [HPY96]. Many precautions are necessary to ensure that the outer as well as inner DFS select processes for expansion in a "compatible manner" (see [HPY96] to know how subtle this can be). To ensure such compatibility, some information pertaining to how processes were expanded during the outer DFS must be conveyed to the inner DFS (typically recorded in state vectors). We conjecture that if, on top of all this, selective state caching is to be supported, additional precautions are needed. In contrast, `Twophase` combines partial order reduction and nested DFS without *any such problems* as in-stack checking is simply absent. In addition, PV provides two different mechanisms for selective state caching that are extremely straightforward to argue correct, as well as implement.

In terms of empirical studies, our results on 16 widely different examples (many coming from outside sources) studied here show that PV can yield a significant advantage over SPIN on most of these examples even without selective state caching or dead variable resetting. When these optimizations are turned on, PV ends up performing better in all cases. In the circular linked list protocol example due to Park and Dill [PD96], PV stores 22% of the states stored by SPIN with partial order reduction enabled and selective state caching disabled (2,243,483 versus 10,078,500 states). With selective state caching and partial order reduction but not dead variable resetting, PV stores 10% of the states[2](1,042,522 states). With dead variable resetting and partial order reduction but not selective state caching, the figures are 11.6% (1,175,491 states), and with all enabled, the figures are 4.7% (477,570 states). In the *leader election* protocol taken from [CGP00], PV stores 67% of the number of states stored by SPIN. The maximum search depth, the number of transitions, the amount of memory consumed, and the run times are as follows. For the circular linked list example, PV uses 1.7% of the memory used by SPIN (30M/1,757M) and generates 0.5% of the search depth (19,745/3,561,948), and for leader election, the figures are 16% (62M/376M) for memory, and 46% (132/281) for depth. In the version of the leader election protocol in the SPIN 3.4.3 distribution, SPIN stores 97 states while PV stores 106 states with partial order reduction alone, and merely 9 states with partial order reduction and a 'save none' option of selective caching discussed later. The `pftp` protocol in this distribution is handled by SPIN in 47,356 states while PV takes 252,531 with only partial order reduction, and 31,514 states with all optimizations.

The original motivations for `Twophase`, the proof of its correctness, as well as performance on some real-world examples have been reported in [Nal98,NG01] as well as briefly in [NG98]. This is the first paper that presents a comprehensive study of PV, and points out, backed by extensive experimental data, why algorithms that implement the proviso without *in-stack* checks must be studied more thoroughly in the model-checking community and used whenever appropriate. We invite the readers to critically examine our results, our examples, as well as the PV code[3].

[2] With selective state caching, not all generated states will be stored.

[3] But for a hashing routine borrowed from an early version of SPIN, PV has been written from scratch, in C, by Nalumasu. Its code has been stable over the last three years.

PV is written in under 5K lines of C, and runs on Solaris and Linux systems. PV's source, algorithms, and data structures have been documented and are available for easy viewing in Hypertext format. PV as well as the examples discussed in this paper are available from our website. We also point out new features of PV in the area of supporting verification of shared memory protocols against formal memory model specifications using *test model-checking* [NGMG98,GMNG98], its enhanced subset of Promela, as well as its implementation. In particular, a user interface called XPV has been implemented by adapting the XSPIN Tcl/Tk code. The version of XPV (called `xpv01`) supports an interface called MPV for shared memory system verification, currently generating *test automata* for Sequential Consistency, PRAM, and read/write orderings. PV has been successfully used in a number of in-house projects, such as the Utah Avalanche processor project [CKK96].

### 1.1 Overview of `Twophase`

Basically, PV effects partial order reductions only when the ample-set size is one. In fact, `Twophase` follows all the conditions [GKPP95,Val96,Pel96b] necessary to preserve the class of CTL$^*$-X assertions. However, the implementation of `Twophase` uses the on-the-fly model-checking algorithm of [CVWY90], and hence handles only LTL-X assertions. During the execution of `Twophase` using depth-first state traversal, whenever a list of states $s_0, \ldots, s_k$, each having a singleton ample-set, are traversed, we say that PV is executing its *phase-1*. During phase-1, it must be ensured that the reduction is not being overly aggressive. That is, a cycle of phase-1 states must not be formed such that the cycle contains a state in which some transition $\alpha$ is enabled, but is never included in $ample(s)$ for any state $s$ on the cycle. This is called the *reduction proviso*, or simply "the proviso." SPIN implements the proviso via an *in-stack* check, as described in [CGP00, Page 155, C3$'$]. However, PV does not implement the proviso in this manner. Instead, it accumulates $s_0, \ldots, s_k$ in an auxiliary list and checks for a revisitation into this auxiliary list[4]. When a revisitation into the auxiliary list occurs, PV switches to another process, does its phase-1, and so on till all processes have been examined for partial order reduction possibilities. It then ends its phase-1 and (i) adds the auxiliary list into the hash table, and (ii) does a phase-2 step where full state expansion is effected. Thereafter, two-phase is recursively applied to every configuration generated by phase-2. Based on the above discussion as well as details to appear in Section 2, we point out two important features of `Twophase`: (i) processes are not left hanging in their penultimate states, but are forced to be reset to their top-level states; (ii) there is no problem combining on-the-fly model checking and partial-order reduction. The main problem with in-stack checking based algorithms is that when the states generated depend on the stack state, it is difficult to guarantee that a given state always generates the same subgraph beneath it whether it is expanded as part of outer DFS or an inner DFS during a nested depth-first search [CVWY90]. However, thanks to the fact that the first phase of `Twophase` does not depend on the stack

[4] Actually, a much cheaper way to effect this check is used, as elaborated later.

state, this guarantee is trivially provided. Also, `Twophase` can be combined easily with selective-caching.

Section 2 elaborates on the `Twophase` algorithm as well as how it is combined with selective caching. Section 3 provides a detailed experimental study. Section 4 elaborates on the implementation details of PV and the version of Promela it supports. Section 5 provides conclusions as well as future plans.

## 2 The `Twophase` Partial Order Reduction Algorithm

### 2.1 Background and Related Work

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. The algorithm in [Pel96a] is discussed in Section 2.2. 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 originally conceived [NG96] 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 [Nal98]. In our past publications [NG97b], we have compared the performance of PV against PO-PACKAGE [God95] on several examples and found that PV outperforms PO-PACKAGE, since the latter is also based on the in-stack check based implementation of the proviso.

We assume a process-oriented modeling language with each process maintaining a set of local variables that it alone can access, global variables that every process can access, and channels that may be tagged as explicit send (`xs`) or explicit receive (`xr`). The set of values of local variables forms the *local state* of a process. For convenience, each process is assumed to contain a distinguished local variable called program counter. 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 process has a program counter ("pc") which 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 the next state 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 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. Some more definitions used in this paper are as follows.

**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 (as far as the properties being verified go), whereas two local transitions of two different processes always 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 partial order reduction.

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.

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.

```
model_check()
{ V_r := φ;  /* Hash table */
  dfs_po(initial_state)
}

dfs_po(s)
{ push(s,stack);
V_r := V_r + {s};
foreach transition t in ample(s)
    if t(s) ∉ V_r then
         dfs_po(t(s));
    endif;
endforeach;
pop(stack);
}
```

```
ample(s) /* refers to stack in dfs_po() */
{ 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) ∈ stack)) 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 found */
 return all enabled transitions;
}
```

**Fig. 1.** In-stack checking based partial order reduction algorithm

### 2.2 In-stack check Based Partial Order Reduction Algorithms

Partial order reduction algorithms that implement the reduction proviso via an *in-stack* check have a pseudo-code similar to that shown in Figure 1 [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 2. Figure 2(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 2(b).

Figure 2(c) shows the state graph generated by the partial order reduction algorithm in Figure 1. 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. $\tau_2$(<s1,s0>) = <s0,s0>, and

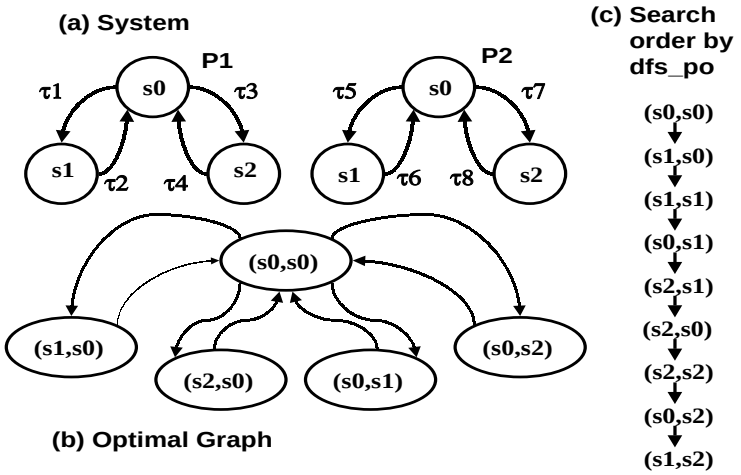


**Fig. 2.** A simple example, its optimal reduced graph, and the reduced graph generated by dfs_po

when dfs_po(<s1,s0>) is called, `stack`={<s0,s0>}. As a result ample(<s1,s0>) cannot return $\{\tau_2\}$; it returns $\{\tau_5, \tau_7\}$. Executing $\tau_5$ from (<s1,s0>) results in <s1,s1>, the third state in the figure. Continuing this way, the graph shown in Figure 2(c) is obtained.

### 2.3 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. We believe that this phenomenon carries over into the models of realistic reactive systems also. For example, in many, or even most, reactive system models, a transaction typically involves a subset of processes. Take 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 in-stack check, 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, which then multiplies out with the states of the remaining processes. This was illustrated by the example in Figure 2. Figure 4 shows the performance obtained by running seven automata similar to P1 of Figure 2 in parallel. SPINunopt corresponds to a run with dataflow optimizations, dead variable resetting, and statement merging turned off. The only purpose of turning off these flags is to show the potential for state-explosion due to the in-stack check. This point will be made again in Figure 5. The optimized SPIN run generates 1 state in this extremely contrived example.

```
model_check()
{ V_r := φ;  /* Hash table */
  Twophase(initial_state);
}
```

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

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

   /* Phase 2: Classic DFS */
   if s∉V_r then
     V_r := V_r + all states in list + {s};
     foreach enabled transition t do
       if t(s) ∉ V_r then
         Twophase(t(s));
       endif;
     endforeach;
   else
     V_r := V_r + all states in list;
   endif;
}
```

**Fig. 3.** The `Twophase` algorithm

We also ran seven instances of another process called `worst.pr` that simply represents a binary fork that dead-ends, as given by the code:

```
active [N] proctype worst()
{ byte b = 1;  if :: b = 2; :: b = 3; fi;  end: 0; }
```

`Twophase` with dead variable resetting optimization turned off will take $3^N$ states as evidenced by the data in our table ($2187 = 3^7$) while SPINunopt will take $2^{(N+1)}$ states as again evidenced by the data in our table. All other SPIN runs reported in this paper are *with* these optimizations activated. Section 3 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 3. 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. Note that `phase1` is *more general* 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 while in `phase1`, actions of multiple processes are executed. Section B provides a correctness sketch of `Twophase`.

| Protocol File Name | Tools Used | States | Hashing | Transitions | Memory | Depth | Time |
|---|---|---|---|---|---|---|---|
| best.pr | PV (no DVR, SaveNone) | 15 | 30 | 28 | 2.15962 | 1 | 0.06 |
| | SPIN | 1 | N/A | 29 | 1.49300 | 0 | 0.06 |
| | SPINunopt | 2187 | N/A | 9517 | 1.49300 | 1457 | 0.12 |
| worst.pr | PV (DVR on) | 128 | 1794 | 896 | 2.36897 | 8 | 8 |
| | PV (DVR off) | 2187 | 20414 | 10206 | 2.41006 | 8 | 17 |
| | SPIN | 8 | N/A | 15 | 2.54200 | 7 | 1 |
| | SPINunopt | 255 | N/A | 255 | 2.54200 | 7 | 2 |

**Fig. 4.** Runs of `best.pr` and `worst.pr`

### 2.4 Selective caching

Under the 'Save All' option of PV, every state `s` is added to `list` in the line marked 1 in `phase1` (and hence also into the main hash table $V_r$ in `Twophase`). This corresponds to selective state caching being turned off. Under the 'Save Backedges' option, not all states are added to `list`. The reason for maintaining `list` is to ensure that the `while` loop terminates. This guarantee can still be provided 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 $<$.) Finally, under the 'Save None' option, `phase1` simply banks on the fact that the deterministic zone of most "typical" processes will eventually end, and adds *no state at all* into `list` at 1. The penalty is that there is a clear risk of looping forever in phase-1. In later sections, we demonstrate that Save None is actually able to handle all our examples without looping, and hence also with maximal savings.

### 2.5 Demonstration of the In-stack Proviso and Selective Caching

Consider the three extremely simple examples referred to as `basic.pr`, `local.pr`, and `global.pr` in Figure 5. Results for these examples also appear in this figure. The captions used in this table are as follows. "SSC" refers to the selective state caching method (with SaveAll indicating that SSC is turned off). "States" refers to the number of states stored by each tool. "Hashing" refers to the number of calls to the hashing function. "Transitions" refers to the total number of transitions generated in searching the graph. "Memory" refers to the total physical memory used. "Depth" refers to the maximum DFS search depth. "Time" refers to CPU time in seconds used for the verification.

In `basic.pr`, SPIN takes 65,793 states to finish the search. This is mainly because of the following reason. Both processes P and Q are eligible for execution. However, moving one process sequentially soon causes the *in-stack* check to succeed, forcing a move of the other process. This repeats till all combinations of `x` and `y` are generated, as can be confirmed by running under SPIN and tracing the variable values. This is clearly not needed: so long as each local variable is individually taken through all its values, all properties with respect to it can be established. Again, not being able to "reset" one process fully seems to be the underlying reason.

```
basic.pr
--------

proctype P()
{ byte x;
  do
  :: x++
  od }

proctype Q()
{ byte y;
  do
  :: y++
  od }

init
{ run P();
  run Q(); }
```

```
local.pr
--------

proctype P()
{ byte x;
  do
  :: x++
  od }

proctype Q()
{ byte y;
  do
  :: y++;
     assert(0)
  od }

init
{ run P();
  run Q(); }
```

```
global.pr
---------

byte y; /* global var */
proctype P()
{ byte x;
  do
  :: x++
  od }

proctype Q()
{
  do
  :: y++;
     assert(0)
  od }

init
{ run P();
  run Q(); }
```

| Protocol File Name | Tools | SSC | States | Hashing | Transitions | Memory | Depth | Time |
|---|---|---|---|---|---|---|---|---|
| basic.pr | PV | SaveAll | 1020 | 1544 | 1538 | 2.27149 | 1 | 0.20 |
| | PV | SaveBackEdge | 1020 | 1544 | 1538 | 2.27149 | 1 | 0.20 |
| | PV | SaveNone | 1 | 1 | 418556112 | 2.26741 | 0 | 90-abort |
| | SPIN | N/A | 65793 | N/A | 66053 | 5.93100 | 65792 | 1.46 |
| | | | | | | | | |
| local.pr | PV | SaveAll | 1 | 1 | 1 | 2.26741 | 0 | 0.10 |
| | PV | SaveBackEdge | 2 | 2 | 1 | 2.24742 | 0 | 0.10 |
| | PV | SaveNone | 1 | 1 | 1 | 2.26741 | 0 | 0.10 |
| | SPIN | N/A | 258 | N/A | 259 | 2.54200 | 257 | 0.09 |
| | | | | | | | | |
| global.pr | PV | SaveAll | 257 | 66307 | 66049 | 2.26844 | 256 | 3.50 |
| | PV | SaveBackEdge | 257 | 66307 | 66049 | 2.26844 | 256 | 2.70 |
| | PV | SaveNone | 1 | 1 | 250639144 | 2.26741 | 0 | 60-abort |
| | SPIN | N/A | 514 | N/A | 516 | 2.54200 | 513 | 0.14 |

**Fig. 5.** In-stack proviso and selective caching variants illustrated

While such sequences of purely local variable accesses do not arise in practice, our later examples tend to show a similar behavior.

In PV, with SaveAll as well as SaveBackEdge, only 1,020 states are generated (thus all combinations of `x` and `y` are not explored). Consider SaveAll for an explanation. `Twophase` selects process `Q` for execution first (the implementation is such that the last process is the first in the scheduler's process list). It keeps executing the `y++` of `Q` without ping-ponging into `P` until the `s` in `list` check in phase-1 of Figure 3 succeeds. It then goes to process `P`, and executes `P` without ping-ponging into `Q`. With SaveNone, PV loops as expected (we abort the run after 90 seconds).

Consider the example `local.pr`. In this, PV finishes even under SaveNone, since it runs process `Q` first, and hence catches the assertion violation. Had we changed the textual order of the processes, `P` would have been run first, causing looping under SaveNone.

The only difference between `global.pr` and `local.pr` is in that `y` is a global variable in the former. `global.pr` works as follows. The two-phase algorithm no longer finds `Q` to have a deterministic move, as `y` is now global. It then moves over to execute `P` in phase-1. Unfortunately, here, SaveNone gets stuck.

In summary, whenever a run with SaveNone terminates, its results are exact. SaveBackEdge gives results close to SaveNone, and SaveAll (selective caching is turned off) is still far more efficient than SPIN's executions on our examples. All our examples terminated under SaveNone.

## 3 Detailed Experimental Evaluation of PV

We have run many examples under SPIN and PV with very minor variations in the input file for syntactic compatibility as explained fully in Section 4. All syntactic variations were to translate `atomic`s into `lock/unlock` of PV and to change the syntax of `run` statements. These differences do not affect the number of states generated. The results appear in Figure 6 and Figure 7 (in log-scale). All the PV runs in this table use the SaveBackEdge option with automatic dead variable resetting. A brief explanation of each example is as follows, with the number of source lines given in parentheses as a crude measure of size.

- `client_server_orig.pr` (105) mimics client/server interactions between two clients and two servers, while `client_server3_orig.pr` (188) is between three clients and servers.
- `inv.pr` (520) and `mig.pr` (326) are directory-based cache coherence protocols proposed for the Utah Avalanche multiprocessor [CKK96].
- `rowo-1.pr` (291) incorporates a highly simplified model of the the HP Runway bus, and verifies the bus for read and write orderings by administering "cleverly constructed" read and write instruction sequences via a non-deterministic automata playing the role of CPUs, as described in [NGMG98].
- `Circular_Linked_List.pr` (257) manages a one-way linked list in which cells may add or remove themselves concurrently as described in [PD96].
- `Leader_Election.pr` (115) is from [CGP00, Page 167].

- All examples under `Partial Store Order` and `Total Store Order` are roughly 140 lines long and test for memory orderings for 1 or 2 addresses with respect to operational definitions of these memory models.
- `pftp.pr` (205) was run straight from the SPIN distribution; in `leader_spin.pr` (89), only the `run` statement was changed.

All of the verification runs, except where noted were run on a dual processor 250MHz Ultra Sparc II machine with 768Meg of physical memory. The Circular Linked List specification (†) required memory resources beyond that available on our dual processor machine. This run, for both tools, was run on a four processor 450MHz Ultra Sparc machine with 4092Meg of physical memory. The SPIN model checker finds a bug in the protocols marked with "*" when there is actually no bug in them. We traced this bug to SPIN's erroneous evaluation of Boolean expressions, and these bugs are being reported. PV evaluates the protocol correctly. We report all flag options used in these runs in Appendix A.1. In particular, SPIN was run with statement merging, while PV does not have this feature available.

To demonstrate the effectiveness of the partial order reduction algorithm relative to selective state caching and dead variable resetting, a permutation of these options have been studied with the results shown in Figure 8.

SSC indicates the use of our selective state caching heuristic. Under SSC, "Yes" means selective state caching with SaveBackEdge, "No" means "SaveAll", and "Unsafe" means "SaveNone". DVR indicates the use of Dead Variable Resetting and automatic resetting with "Yes" meaning the use of this facility and "No" meaning otherwise. All of the verification runs were performed on a dual processor 250MHz Ultra Sparc II machine with 768Meg of physical memory. Flag options are in Section A.2.

## 4 Implementation Details of PV

PV uses an extended subset of Promela. One main reason for the subsetting was to built a functional model checker to test the effectiveness of `Twophase`, without the overhead of building a model checker fully compatible with the existing Promela definition. Also, we did not understand certain constructs such as `atomic` and `timeout` as well as we had hoped for. PV supports enumerated types defined using `typedef enum a, b, c, d T;`. In addition to never claims, invariants are supported. Invariants are specified as `invariant INVARIANT_NAME expression`.

More specifically, PV does not support the following constructs: `else`, `timeout`, synchronous channels (channels with zero size), `atomic/d_step`, and `mtype`. SPIN provides an implicit parameter `_pid`, while PV does not. PV supports `locks` as well as `semaphores` in lieu of `atomics`. While not as general as `atomics`, we find `locks` as well as `semaphores` to be simpler as well as more suited to our domain of examples. `lock`, and `sema` are two new data types. The difference is that semaphore is the classic counting semaphore with operations `lock()` and `unlock()` (which are sometimes referred to as `P()` and `V()` in literature). `Lock`, on the other hand, is a binary semaphore with the restriction that `unlock()` can be done by only the process that currently holds the lock.

| Protocol File Name | Tool | States | Hashing | Transitions | Memory | Depth | Time |
|---|---|---|---|---|---|---|---|
| client_server_orig.pr (a) | PV | 79 | 222 | 240 | 2.30997 | 17 | 0.2 |
| | SPIN | 76 | N/A | 116 | 2.54200 | 75 | 0.1 |
| client_server3.orig.pr (b) | PV | 1411 | 5402 | 6768 | 2.44607 | 162 | 1.1 |
| | SPIN | 1947 | N/A | 4281 | 2.74700 | 1272 | 0.3 |
| inv.pr (c) | PV | 27680 | 60470 | 101184 | 3.09183 | 1094 | 4.6 |
| | SPIN | 388034 | N/A | 722467 | 52.92700 | 105603 | 23.3 |
| mig.pr (d) | PV | 12054 | 24686 | 44248 | 2.63867 | 639 | 1.8 |
| | SPIN | 24410 | N/A | 47676 | 4.38500 | 5858 | 1.6 |
| rowo-1.pr (e) | PV | 2412 | 3870 | 11699 | 2.47784 | 131 | 0.9 |
| | SPIN | 192336 | N/A | 278179 | 62.56000 | 11182 | 24.2 |
| Circular_Linked_List.pr (f) † | PV | 477570 | 1723268 | 1487501 | 29.12370 | 19745 | 271.8 |
| | SPIN | 10078500 | N/A | 22312700 | 1757.42700 | 3561948 | 1281.6 |
| Leader_Election.pr (g) | PV | 684112 | 6252358 | 7797612 | 62.24800 | 132 | 2460.0 |
| | SPIN | 1016380 | N/A | 3623060 | 376.93100 | 281 | 714.6 |
| Leader_spin.pr (h) | PV | 26 | 29 | 105 | 2.32907 | 5 | 0.1 |
| | SPIN | 91 | N/A | 91 | 2.54200 | 102 | 0.1 |
| pftp.pr (i) | PV | 31964 | 87842 | 98498 | 4.59044 | 439 | 13.0 |
| | SPIN | 47356 | N/A | 64970 | 8.58400 | 1923 | 3.6 |
| Partial Store Order | | | | | | | |
| (CMP,POS)_2_address.pr (j) | PV | 1037 | 15008 | 5002 | 2.38997 | 465 | 1.6 |
| | SPIN | 2362 | N/A | 9576 | 2.64400 | 510 | 0.4 |
| Total Store Order | | | | | | | |
| (CMP,POS)_1_address.pr | PV | 102 | 656 | 327 | 2.30866 | 16 | 0.2 |
| | SPIN | 106 | N/A | 332 | 2.54200 | 37 | 0.1 |
| (CMP,POS)_2_address.pr (k) | PV | 1260 | 11468 | 3822 | 2.33658 | 319 | 1.1 |
| | SPIN | 2987 | N/A | 9157 | 2.64400 | 261 | 0.2 |
| (CMP,POS,WA)_1_address.pr | PV | 676 | 5432 | 1810 | 2.31710 | 50 | 0.5 |
| | SPIN | 680 | N/A | 1815 | 2.54200 | 75 | 0.1 |
| (CMP,POS,WA)_2_address.pr* (l) | PV | 23253 | 202958 | 67652 | 2.91780 | 523 | 16.3 |
| | SPIN | 51141 | N/A | 154687 | 5.00000 | 645 | 7.3 |
| (CMP,RO,WOS)_1_address.pr (m) | PV | 102 | 866 | 288 | 2.30866 | 16 | 0.2 |
| | SPIN | 106 | N/A | 293 | 2.54200 | 37 | 0.1 |
| (CMP,RO,WOS)_2_address.pr* (n) | PV | 2345 | 20729 | 6909 | 2.36286 | 435 | 1.4 |
| | SPIN | 7577 | N/A | 23854 | 2.84900 | 529 | 1.5 |

**Fig. 6.** PV runs with SaveBackEdge and Dead Variable Resetting versus SPIN

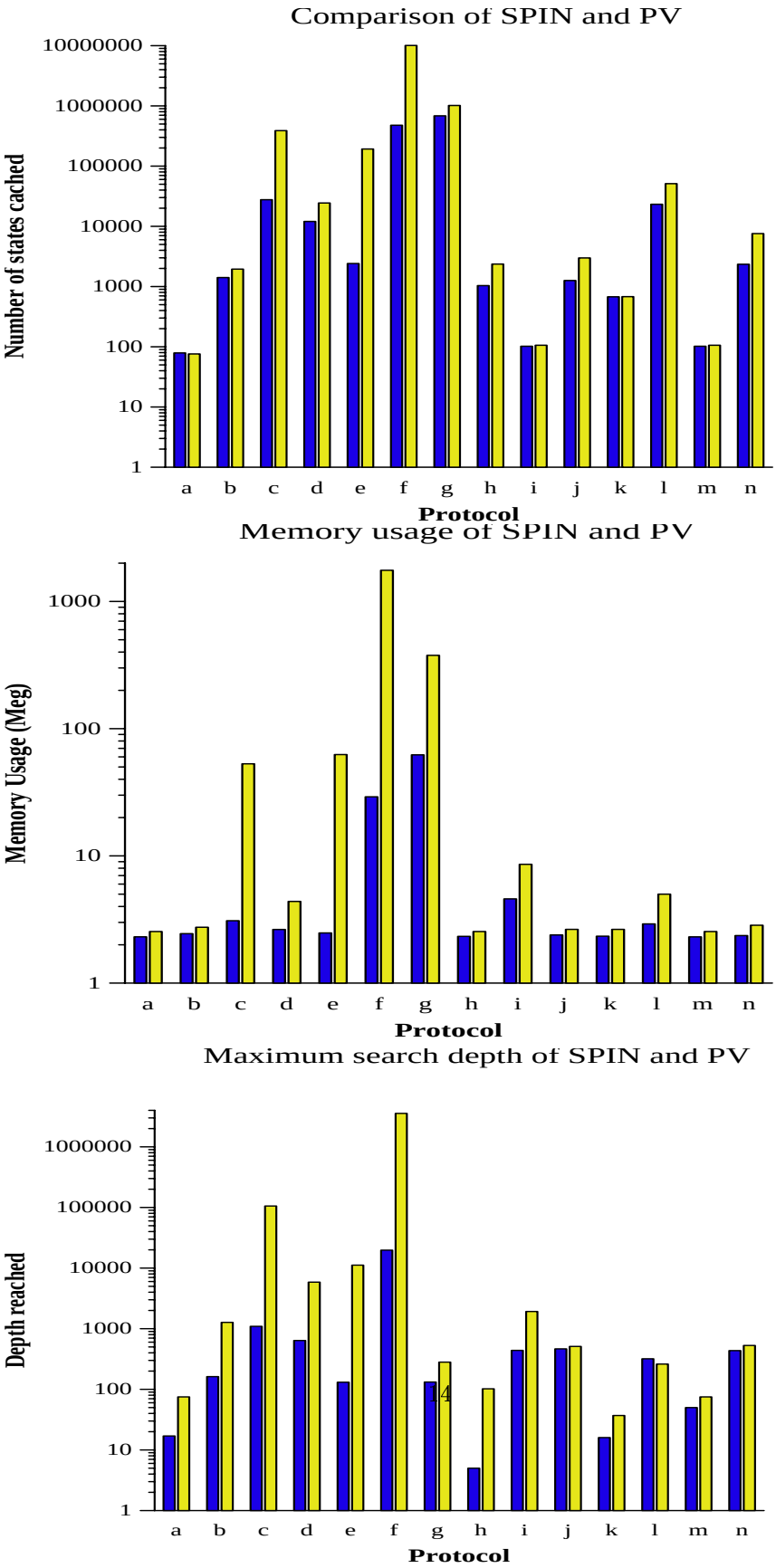


**Fig. 7.** Log graphs comparing PV (left bars) and SPIN (right bars) on States, Memory, and Search Depth

| Protocol File Name | Tool | SSC | DVA | States | Hashing | Transitions | Memory | Depth | Time |
|---|---|---|---|---|---|---|---|---|---|
| client_server_orig.pr | PV | SaveBackEdge | Yes | 79 | 222 | 240 | 2.30997 | 17 | 0.1 |
| | PV | SaveBackEdge | No | 79 | 222 | 240 | 2.30997 | 17 | 0.1 |
| | PV | SaveAll | Yes | 117 | 326 | 240 | 2.31115 | 17 | 0.1 |
| | PV | SaveAll | No | 117 | 326 | 240 | 2.31115 | 17 | 0.1 |
| | PV | SaveNone | Yes | 77 | 170 | 240 | 2.30991 | 17 | 0.1 |
| | SPIN | N/A | N/A | 76 | N/A | 116 | 2.54200 | 75 | 0.2 |
| client_server3.orig.pr | PV | SaveBackEdge | Yes | 1411 | 5402 | 6768 | 2.44607 | 162 | 0.9 |
| | PV | SaveBackEdge | No | 1411 | 5402 | 5402 | 2.44607 | 162 | 0.9 |
| | PV | SaveAll | Yes | 2296 | 8606 | 6768 | 2.51090 | 162 | 1.5 |
| | PV | SaveAll | No | 2296 | 8606 | 6768 | 2.51090 | 162 | 1.3 |
| | PV | SaveNone | Yes | 1360 | 3674 | 6768 | 2.44281 | 162 | 1.2 |
| | SPIN | N/A | N/A | 1947 | N/A | 4281 | 2.74700 | 1272 | 0.3 |
| inv.pr | PV | SaveBackEdge | Yes | 27680 | 60470 | 101184 | 3.09183 | 1094 | 4.9 |
| | PV | SaveBackEdge | No | 79912 | 167001 | 280495 | 4.56318 | 2472 | 13.2 |
| | PV | SaveAll | Yes | 60736 | 127670 | 101184 | 4.02774 | 1094 | 8.6 |
| | PV | SaveAll | No | 170138 | 354147 | 280495 | 7.11293 | 2472 | 21.8 |
| | PV | SaveNone | Yes | 24626 | 52970 | 101184 | 3.00605 | 1094 | 4.1 |
| | SPIN | N/A | N/A | 388034 | N/A | 722467 | 52.92700 | 105603 | 23.3 |
| mig.pr | PV | SaveBackEdge | Yes | 12054 | 24686 | 44248 | 2.63867 | 639 | 1.9 |
| | PV | SaveBackEdge | No | 12054 | 24686 | 44248 | 2.63867 | 639 | 2.2 |
| | PV | SaveAll | Yes | 28580 | 55114 | 44248 | 3.10026 | 639 | 3.5 |
| | PV | SaveAll | No | 28586 | 55114 | 44248 | 3.10038 | 639 | 4.0 |
| | PV | SaveNone | Yes | 10304 | 21730 | 44248 | 2.59548 | 639 | 1.8 |
| | SPIN | N/A | N/A | 24410 | N/A | 47676 | 4.38500 | 5858 | 1.6 |
| rowo-1.pr | PV | SaveBackEdge | Yes | 2412 | 3870 | 11699 | 2.47784 | 131 | 0.8 |
| | PV | SaveBackEdge | No | 6375 | 10180 | 31622 | 2.76256 | 277 | 2.0 |
| | PV | SaveAll | Yes | 9760 | 13366 | 11699 | 3.00175 | 131 | 1.9 |
| | PV | SaveAll | No | 24486 | 36014 | 31622 | 4.05224 | 277 | 6.0 |
| | PV | SaveNone | Yes | 2352 | 3332 | 11699 | 2.47400 | 131 | 1.0 |
| | SPIN | N/A | N/A | 192336 | N/A | 278179 | 62.56000 | 11182 | 24.2 |
| Circular_Linked_List.pr | PV | SaveBackEdge | Yes | 477570 | 796740 | 1487501 | 30.88370 | 19745 | 48.3 |
| | PV | SaveBackEdge | No | 1042522 | 1723268 | 3377565 | 63.05100 | 58302 | 107.2 |
| | PV | SaveAll | Yes | 1175491 | 1885872 | 1487501 | 69.57070 | 19745 | 98.3 |
| | PV | SaveAll | No | 2243483 | 4239200 | 3377565 | 130.00000 | 58302 | 226.0 |
| | PV | SaveNone | Yes | 477570 | 796740 | 1487501 | 30.8837 | 19745 | 51.6 |
| | SPIN | N/A | N/A | 10078500 | N/A | 22312700 | 1757.42700 | 3561948 | 1189.2 |
| Leader_Election.pr | PV | SaveBackEdge | Yes | 684112 | 5568246 | 7797612 | 62.24800 | 132 | 690.0 |
| | PV | SaveBackEdge | No | 684112 | 5568246 | 7797612 | 62.24800 | 132 | 684.7 |
| | PV | SaveAll | Yes | 1134651 | 10923792 | 7797612 | 101.41300 | 132 | 1020.2 |
| | PV | SaveAll | No | 1134651 | 10923792 | 7797612 | 101.41300 | 132 | 1041.9 |
| | PV | SaveNone | Yes | 684112 | 6252358 | 7797612 | 62.24800 | 132 | 675.0 |
| | SPIN | N/A | N/A | 1016380 | N/A | 3623060 | 376.93100 | 281 | 714.6 |
| Leader_spin.pr | PV | SaveBackEdge | Yes | 26 | 29 | 105 | 2.32907 | 5 | 0.1 |
| | PV | SaveBackEdge | No | 26 | 29 | 105 | 2.32907 | 5 | 0.1 |
| | PV | SaveAll | Yes | 106 | 111 | 105 | 2.33279 | 5 | 0.1 |
| | PV | SaveAll | No | 106 | 111 | 105 | 2.33279 | 5 | 0.1 |
| | PV | SaveNone | Yes | 9 | 10 | 105 | 2.32826 | 5 | 0.1 |
| | SPIN | N/A | N/A | 91 | N/A | 91 | 2.54200 | 108 | 0.1 |
| pftp.pr | PV | SaveBackEdge | Yes | 31964 | 87842 | 98498 | 4.59044 | 439 | 13.0 |
| | PV | SaveBackEdge | No | 139186 | 421904 | 487545 | 12.29940 | 643 | 61.3 |
| | PV | SaveAll | Yes | 61774 | 129711 | 98498 | 6.70826 | 439 | 17.8 |
| | PV | SaveAll | No | 252531 | 638043 | 487545 | 20.37850 | 643 | 87.7 |
| | PV | SaveNone | Yes | 31514 | 62424 | 98498 | 4.55474 | 439 | 10.0 |
| | SPIN | N/A | N/A | 47356 | N/A | 64970 | 8.58400 | 1923 | 3.6 |

**Fig. 8.** Relative Effect of the Optimizations

In PV, run statements may appear only within `init` sections. Further, `init` sections can contain only run statements. To run multiple instances of a process, we allow the iterator construct shown below:

```
run [i1<n1][i2<n2][i3<n3]... proc_name(parameters)
```

For instance, the `run` statement used in `leader_spin.pr` was

```
run[proc<N] node (q[proc], q[(proc+1)%N], (N+I-proc-1)%N+1)
```

Here, `n1, n2, n3` are compile time evaluable constants. The `parameters` may involve the variables `i1, i2, i3, ....`

PV uses `cc -E` to implement the C preprocessor functionality. A different preprocessor can be specified via option -c, for example `"-c/usr/lib/cpp"`. This preprocessor must remove the comments from the file (which all C preprocessors do by default). PV defines a new pre-processors symbol `_UV_`. With this, one can write files that work with both PV and SPIN. Most of our experiments reported here were performed based on common source Promela programs with differences customized using the `_UV_` flag. In some cases, distinct, but only slightly different, source files were used.

An example of simulating `lock` and `unlock` in SPIN is as follows (differences in the `init` section can also be similarly treated):

```
-- declarations
#ifdef _UV_
# define LOCK lock
#else

/* Simulate 'lock' in SPIN */
# define LOCK bit
# define lock(x) atomic{x==0 -> x=1;}
# define unlock(x) x = 0;
#endif
```

```
-- usage

LOCK l;
proctype p() {
    ...
  lock(l);
  unlock(l);
   ...  }
```

## 5 Conclusions

In this paper, we present a thorough case study of our enumerative model-checker PV that is similar to SPIN in many ways. PV is based on our partial order reduction algorithm called `Twophase` that implements the ample set calculation and the proviso condition that prevents the ignoring problem without using the in-stack check that is currently the most widely used method. By design, `Twophase` guarantees that the state graph starting at a given state is the same in `dfs1` and `dfs2` of the on-the-fly LTL-X checking algorithm of [CVWY90]. This makes the combination of `Twophase` with on-the-fly model-checking as well as selective caching considerably simpler in PV. PV also supports several variants of selective caching, supports automatic dead variable resetting, and a user interface geared towards verifying shared memory systems against formal memory models. PV accepts an enhanced subset of Promela, is well documented through a hypertext document, has been in

active use over the past four years, and is available along with all examples reported here from our website. On our list of examples, PV offers significant state, memory, runtime, and search-depth advantages over SPIN. Many more examples were run over the past four years, and in a majority of them, PV outperformed SPIN. We hope that these results will prompt a critical study of PV by other groups, prompt its wider use, and lead to the incorporation of our ideas (or source code) in other tools. One critical enhancement to be soon undertaken is support for the full Promela language, so that we may freely run many existing SPIN benchmarks that we cannot currently run. We also plan to add bit-state hashing as well as enhance the user interface of PV and XPV considerably. Several new features PV are under consideration, especially in the area of formal verification of conformance to shared memory models.

## References


[CGP00] Edmund Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, 2000.

[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.

[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.

[GKPP95] R. Gerth, R. Kuiper, W. Penczek, and Doron 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.

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

[HPY96] Gerard J. Holzmann, Doron 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.

[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.

[NG98] 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.

[NG01] Ratan Nalumasu and Ganesh Gopalakrishnan. An efficient partial order reduction algorithm with an alternative proviso implementation. *Formal Methods in System Design*, 2001. To Appear.

[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.

[PD96] S. Park and D. L. Dill. Protocol verification by aggregation of distributed transactions. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 300–309, New Brunswick, NJ, USA, July 1996. Springer-Verlag.

[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.

[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.

# A Flags and other Options

## A.1 Basic Runs

- The flags used with PV and SPIN for various runs are as follows. To generate the C code, we used -X -a.
- The flags used to compile each of the PV runs, with the exception of the rowo-1.pr, specification were:
  -DSAVE_BACK_EDGE -DPOSIX_SOURCE -DELIM_DEAD -DALGORITHM1 -DREDUCE -DNOFAIR.
  The rowo-1.pr specification required the addition of the -DSKIP_XRXS flag in both PV and SPIN, as this usage was determined using auxiliary reasoning to be safe.
- The flags used to compile each of the SPIN runs were
  -_DPOSIX_SOURCE -DMEMLIM=mem -DSAFETY -DNOCLAIM -DXUSAFE -DNOFAIR
  where mem indicates the physical memory limit specified at compile time. PV takes this parameter at run time.
- Run time flags for SPIN and PV varied as necessary to complete the various verification run. PV's flags are as follows:
  -d(maximum allowable search depth) -m(physical memory limit)M -h(hash table size) -e1.
  SPIN's flags are: -m(maximum allowable search depth) -w(hash table size) -A -c1.
- The maximum allowable search depth was adjusted up to be within the nearest factor of 10 for each verification run (this makes SPIN's memory usage statistics quite accurate).
- The default hash table parameter setting of SPIN (19) and of PV (18) were used.

## A.2 SSE and DVR flags

The compilation flags differed depending upon the option desired. All of the verifications were run with the same flags, except for rowo-1.pr which also requires the -DSKIP_XRXS flag. The permutation worked out as follows:

- SSC and DVR:
  -DSAVE_BACK_EDGE -DELIM_DEAD -DPOSIX_SOURCE -DALGORITHM1 -DREDUCE -DNOFAIR
- SSC only:
  -DSAVE_BACK_EDGE -DPOSIX_SOURCE -DALGORITHM1 -DREDUCE -DNOFAIR
- DVR only:
  -DSAVE_ALL -DELIM_DEAD -DPOSIX_SOURCE -DALGORITHM1 -DREDUCE -DNOFAIR
- Without SSC and DVR:
  -DSAVE_ALL -DPOSIX_SOURCE -DALGORITHM1 -DREDUCE -DNOFAIR

  The run time flags for the PV system are as shown above.

## B A Correctness sketch of `Twophase`

The term *proviso* is used to refer to condition Ä5 of [Val96, Page 222], 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 Ä5 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.

The correctness of `Twophase` follows from previous results. In particular, Theorem 6.3 of [Val96] states that if conditions Ä5 and Ä8 hold, the reduced and the unreduced transition systems are branching-bisimilar. Here, condition Ä5 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 3 under `Twophase` records all those states that are fully expanded). Condition Ä8 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 5. 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].