PV: A Model-Checker for Verifying LTL-X Properties

Ratan Nalumasu       Ganesh Gopalakrishnan

Department of Computer Science
University of Utah, Salt Lake City, UT 84112
{ratan, ganesh}@cs.utah.edu
February 28, 1997

\[\text{Supported in part by ARPA Order \#B990 Under SPAWAR Contract \#N0039-95-C-0018 (Avalanche), DARPA under contract \#DABT6396C0094 (UV), and NSF MIP MIP-9321836.}\]
Abstract

We present a verification tool $PV$ (Protocol Verifier) that checks stutter-free LTL (LTL-X) properties using a new partial order reduction algorithm called Two phase. Two phase significantly reduces space and time requirements on many practically important protocols on which the partial order reduction algorithms implemented in previous tools [God95, HP94, Pel96] yield very little savings. In some cases, such as one version of the invalidate directory protocol of the Utah Avalanche multiprocessor, current tools did not finish their search while Two phase finished and discovered a bug that was missed by the unfinished runs of existing tools. Two phase's performance is largely due to the fact that it does not employ a run-time proviso such as used in other tools.

In [NG97], we motivated the problems caused by the proviso, presented Two phase, and proved that it preserves stutter-free safety properties. In this paper, we provide a proof that the Two phase algorithm also preserves all stutter-free LTL (LTL-X) properties. We also characterize the type of protocols that benefit from the Two phase search strategy and provide experimental evidence showing the advantages of its search strategy.

Keywords: Finite system verification, Explicit enumeration, Partial order reductions

1 Introduction

With the increasing scale of hardware systems and the corresponding increase in the number of concurrent protocols involved in their design, formal verification of concurrent protocols is an important practical need. Explicit state enumeration methods [CES86, Hol91, Dil96] have shown considerable promise in verification of real-world protocol verification problems and have been used with success on many industrial designs [YGM+95]. Using most explicit state enumeration tools, a concurrent system is modeled as a set of concurrent processes communicating via shared variables [Dil96] and/or communication channels [Hol91] executing under an interleaving model. An important run-time optimization called partial-order reductions [Pel96, God95, Val93] helps avoid having to examine all possible interleavings among processes, and is crucial to handling large models.

In our research in system-level hardware design, specifically in the verification of cache coherence protocols used in the Utah Avalanche multiprocessor [CKK96], we observed that existing tools that support partial-order reductions [God95, HP94] failed to provide sufficient reductions. We traced this state explosion to their use of run-time provisos (explained later) in deciding which processes to run in a given state. This paper presents a new partial-order reduction algorithm called Two phase that, in most cases, outperforms all comparable algorithms, and is part of a new protocol verification tool called PV that finds routine application in our multiprocessor design project [CKK96]. In some cases (e.g., the invalidate protocol considered for use in the Avalanche processor), not only did PV's search finish when others' didn't, but it also found some bugs which the others missed in their incomplete search. In an earlier paper [NG97], we showed that Two phase preserves safety properties. In this paper, we prove that Two phase preserves all stutter-free LTL (LTL-X) properties. We also provide experimental results of running PV on a number of examples.
2 Background

Two phase and other partial order reduction algorithms depend on commutativity of transitions in the system to reduce the time and memory requirements of the verification job. Commutativity depends on the communication primitives provided. Typical communication primitives include shared memory (global variables), bounded buffers, unbounded buffers, and rendezvous communication. The PV tool \cite{NG96} supports communication through shared memory and bounded buffers. In this paper, for the sake of simplicity, we assume that there is exactly one shared variable (implicitly named $g$), and that there is no other means of communication. The Two phase algorithm and the proofs presented in this paper can be easily modified to work with multiple global variables and bounded buffers.

A process $P_i$ is a tuple $(Q_i, \delta_i, g, q_{init}^i, \Sigma, \sigma_{init})$ where $Q_i$ is a finite set of states that $P_i$ can assume, $q_{init}^i \in Q_i$ is the initial state of $P_i$, $\Sigma$ is the range of an implicit global variable $g$, and $\sigma_{init} \in \Sigma$ is the initial value of $g$. $\delta_i$ is a relation from $Q_i$ to $Q_i$ indicating a set of local transitions of $P_i$, i.e., the set of transitions that do not depend on $g$ in any way. The relation $\delta^g$ from $Q_i \times \Sigma$ to $Q_i \times \Sigma$ is a set of global transitions, i.e., the set of transitions of $P_i$ that read/write the global variable $g$. We also assume that the domains of $\delta_i$ and $\delta^g$ are disjoint and that union of the two domains is $Q_i$\footnote{Domains of $\delta_i$ and $\delta^g$ are disjoint means that if in a guarded command, for example, some of the guards access $g$ and other guards do not, then we act as though all guards access $g$.}. For convenience, we write $(q, \sigma) \xrightarrow{\delta^g} (q', \sigma')$ to indicate that (a) $q \xrightarrow{i} q'$ and $\sigma = \sigma'$ or (b) $(q, \sigma) \xrightarrow{\delta^g} (q', \sigma')$.

As mentioned earlier, partial order reductions depend on the notion of commutativity of transitions. For example, transitions in $\delta_i$ commute with transitions in $\delta^g (j \neq i)$.

**Definition 1 (Local)** A state $q \in Q_i$ is said to be local if it is in the domain of $\delta_i$, i.e., there is at least one $q'$ such that $q \xrightarrow{i} q'$. When $P_i$ is in a local state, all of its transitions commute with transitions of other processes.

**Definition 2 (Deterministic)** A state $q \in Q_i$ is said to be deterministic if it is local and the image of the $q$ on $\delta_i$ is singleton, i.e., if $q \xrightarrow{i} q'$ and $q \xrightarrow{i} q''$ then $q' = q''$. In this case, refer to $q'$ as $\texttt{next}(q)$. While other algorithms depend on the notion of local to reduce the graph, Two phase algorithm uses deterministic.

To find the full state space generated by interleaved execution of processes $P_1, P_2, \cdots, P_n$, referred to as the graph $G_T = (V_T, E_T)$, following graph traversal algorithm can be used:

- $(q_{init}^1, q_{init}^2, \cdots, q_{init}^n, \sigma_{init}) \in V_T$,
- if $Q = (q_1, q_2, \cdots, q_i, \cdots, q_n, \sigma) \in V_T$ and $(q_i, \sigma) \xrightarrow{i} (q_i', \sigma')$ then $Q' = (q_1, q_2, \cdots, q_i', \cdots, q_n, \sigma') \in V_T$ and $(Q, Q') \in E_T$. ($Q'$ is called a $P_i$ successor of $Q$).

Definitions of local and deterministic are extended to a concurrent system as follows:
Definition 3 (Local) A process $P_i$ is said to be local in state $(q_1, q_2, \ldots, q_i, \ldots, q_n, \sigma)$ iff $q_i$ is local in $P_i$.

Definition 4 (Deterministic) A process $P_i$ is deterministic in $Q = (q_1, q_2, \ldots, q_i, \ldots, q_n, \sigma)$ iff $q_i$ is deterministic in $P_i$. In this case, we write deterministic($Q,i$). In this case, we use $next(i,Q)$ to indicate the state generated by moving $P_i$, i.e., if $next(q_i) = q'_i$, $next(i,Q) = (q_1, q_2, \ldots, q'_i, \ldots, q_n, \sigma)$.

Note that if a process $P_i$ is local in $Q$, then executing a process $P_j$ ($j \neq i$) does not affect $P_i$ in anyway.

To simplify presentation of the algorithm and proofs, we use the following notation. We write “$Q \rightarrow Q'$” to indicate that $(Q, Q') \in E_f$ and “$Q \rightarrow$” to indicate the set $\{Q'|(Q, Q') \in E_f\}$. Similarly, we write $\ll Q_0, Q_1, \ldots, Q_n \gg E$ to indicate that $(Q_i, Q_{i+1}) \in E$ for $i \in \{0..n-1\}$. If $Q = (q_1, q_2, \ldots, q_i, \ldots, q_n, \sigma)$, $Q_i(Q) = q_i$ and $\Sigma(Q) = \sigma$.

Definition 5 (Property) We are interested in preserving the truth value all LTL-X properties involving only the global variable $g$.

Related Work

To determine whether a given property of the above form is true or not, many times it is not necessary to construct the entire $G_f$. Partial order reductions attempt to generate a subgraph of $G_f$, called reduced graph $G_r$, that satisfies the property iff the property is satisfied by $G_f$. As mentioned earlier, these algorithms [Val93, NG97, God95, Pel96] attempt to generate $G_r$ by exploiting the fact that when a process $P_i$ is in a local state, its transitions commute with the transitions of $P_j$ ($j \neq i$). When $P_i$ is local from a state $S$ that is about to be expanded by the graph generating algorithm, the partial order algorithm may, for example, select to expand transitions of process $P_i$ only, and postponing transitions of all other processes. Of course, special care must be taken to ensure that no enabled transition is indefinitely postponed.

Previous algorithms such as [God95] and [Pel96] use proviso to ensure that no transition is indefinitely postponed. Both these algorithms (as well as Two Phase) use a stack to maintain the list of states currently being expanded. When expanding the current top of the stack, [God95] and [Pel96] algorithms require that the subset of transitions selected to explore do not result in a state that is already on the stack. Usage of the proviso, in some cases, causes the $G_r$ to be quite large even though a much smaller $G_r$ can be computed[NG97]. Two phase algorithm attempts to rectify this problem by avoiding the proviso and using a different search strategy.
3 Two Phase Algorithm

Two phase is shown in Figure 1. Unlike previous algorithms, this algorithm does not use proviso. Instead its execution is divided into two phases. In phase I, the algorithm executes transitions of deterministic processes. In this phase, the algorithm maintains a list of states visited in variable list. Without this variable, if a process is in a deterministic loop, the

\[2\text{The provisos differ slightly depending on whether they preserve LTL-X or safety only.}\]

\[3\text{The proviso used in the two algorithms differ slightly because [God95] preserves only stutter-free safety properties while [Pel96] preserves LTL-X. However, the effect of proviso is very similar on performance of both algorithms.}\]

init stack to contain initial state
init \( V_r \) to \( \Phi \)
init \( E_r \) to \( \Phi \)

Two-phase()
{
  s := top(stack);
  list := \{s\};
  /* Phase I: partial order step */
  for i := 1 to nprocesses {
    while (deterministic(s,i)) {
      /* Execute the only enabled transition of \( P_i \) */
      \( E_r := E_r \cup \{(s, next(i,s))\} \);
      s := next(i, s);
      if \( (s \not\in \text{list}) \) goto NEXT_PROC;
      \( \text{list := list + \{s\};} \);
    }
    \( \text{NEXT_PROC: */ } \)
    \( \text{next i */} \)
  }
  /* Phase II: classical DFS */
  if \( (s \not\in E_r) \) {
    \( V_r := V_r \cup \text{list;} \)
    for each \( \text{succ in } s \rightarrow \) {
      if \( (\text{succ} \in E_r) \{
        \text{push(succ, stack);}
        \text{Two-phase();}
      \}
    }
    assert(\( s \rightarrow \subseteq V_r \));
    \( E_r := E_r \cup \{(s,t) \mid s \rightarrow t\}; \)
  } else {
    \( V_r := V_r \cup \text{list;} \)
  }
  pop (stack);
}

Figure 1: Two-phase() avoids proviso using a different execution strategy.
while loop in the first phase would not terminate (see Lemma 2.) In the second phase, list is added to the $V_r$. In phase II, the algorithm first checks if $s$ is already in $V_r$. If it is not in $V_r$, $s$ is fully expanded (lines 18–23). If $s$ is already in $V_r$, then either $s$ is already expanded or a deterministic successor of $s$ is expanded. Hence, it is not necessary to explore $s$ (line 27) and the recursive call terminates.

As an example, consider the protocol shown in Figure 2(a). On this protocol, the state space generated by [Pel96] algorithm, implemented in [HP96], is shown in Figure 2(c). As can be seen, when the search reaches the state $<S_0, S_1>$, due to proviso, the algorithm selects process P2 and generates $<S_1, S_1>$. In fact, since state $<S_0, S_0>$ is in the stack until the entire graph generation is completed, proviso is invoked many times, causing the algorithm to generate all states in the system (though not all edges).

Two phase avoids this problem by completely avoiding the proviso. Instead, it depends on deterministic states such as $S_1$ and $S_2$ in P1 and P2 to bring the reductions. The graph constructed by the Two phase algorithm on the same protocol is shown in Figure 2(c). In this protocol, the initial state $<S_0, S_0>$ is not deterministic with respect to either of the processes. Hence no states are generated in Phase I of the algorithm. In phase II, $<S_0, S_0>$ is completely expanded, resulting in four states $<S_1, S_0>, <S_2, S_0>, <S_0, S_1>$, and $<S_0, S_2>$. Then Two-phase is called on all these four states. Let us consider the case of expanding $<S_1, S_0>$. Expansion of other states is similar. Process P1 is deterministic in $<S_1, S_0>$, and hence P1 is executed in phase I, resulting in $<S_0, S_0>$. At this point because neither P1 nor P2 is deterministic in $<S_0, S_0>$. In phase II, the algorithm would discover that $<S_0, S_0>$ is already in $V_r$, and the recursive call terminates, resulting in the state graph shown in Figure 2(b). As can be seen, Two phase generates a much smaller graph, as a direct result of avoiding the proviso.

In Figure 1 all intermediate states are added to list. Note that the only reason for maintaining list is to ensure that the while loop in Phase I terminates. However, we can add only a subset of states to list and still guarantee that the while loop terminates. For example, one can add $s$ to list only when the new value of $s$ is bit-wise smaller than the value of old $s$ (i.e., value of $s$ before executing $s := \text{next}(i,s)$). In this case, of course, line 8 also has to be modified appropriately. This technique constitutes a simple form of selective caching. PV also supports this selective caching.

4 Experimental Results

The Two phase algorithm outperforms the [Pel96] algorithm and a similar algorithm implemented in PO-PACKAGE$^5$[God95] when the proviso is invoked often. 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

$^4$Adding list to the $V_r$ is really not necessary, but for our implementation it is easier to add list to $V_r$, than to reset list.

$^5$PO-PACKAGE uses a weaker form of proviso along with sleepsets [GHP92], but it preserves only safety properties.
the proviso, state resetting cannot be done as the initial state will be in the stack until the entire reachability analysis is completed. Since at least one process is not reset, the algorithm generates unnecessary states, thus increasing the number of states visited. In other realistic systems also the number of extra states generated due to the proviso can be high. Two phase does not use the proviso, thus avoiding generating the extra states.

Table 1 shows results of running the [Pel96] algorithm implemented in SPIN [HP96], Two phase with selective state caching disabled, and Two phase with selective caching enabled on various protocols. This table shows number of states in $V_q$ and time taken in seconds to complete the graph construction on a Super Sparc 20. All verification runs are limited to 64MB so that the entire graph would fit in physical memory. Protocols B5–B7 are best case protocol in Figure 2 with $N=5$, 6, and 7. Protocol W5–W7 is example of a protocol that runs better with SPIN search algorithm. On this protocol, the proviso is never invoked, and this protocol has no deterministic states. As a result the Two phase degenerates to full state space, while SPIN reduces the number of states appreciably (from $3^n$ states to $2^{n+1} - 1$ states for where $n=5$, 6, or 7).

*Mig* and *inv* are two cache coherency protocols used in [CKK96]. On *inv*, SPIN fails to complete the graph construction in 64MB of memory. PV tool on the other hand finishes comfortably generating 255,781 states (without selective caching) or 135,404 states (with selective caching). *SC* is a server/client protocol. This protocol consists of $n$ servers and $n$
<table>
<thead>
<tr>
<th>Protocol</th>
<th>SPIN</th>
<th>PV</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>all</td>
<td>selective</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B5</td>
<td>243/0.34</td>
<td>11/0.33</td>
<td>1/0.3</td>
<td></td>
</tr>
<tr>
<td>B6</td>
<td>729/0.38</td>
<td>13/0.33</td>
<td>1/0.3</td>
<td></td>
</tr>
<tr>
<td>B7</td>
<td>2187/0.50</td>
<td>15/0.33</td>
<td>1/0.3</td>
<td></td>
</tr>
<tr>
<td>W5</td>
<td>63/0.33</td>
<td>243/0.39</td>
<td>243/0.3</td>
<td></td>
</tr>
<tr>
<td>W6</td>
<td>127/0.39</td>
<td>729/0.49</td>
<td>729/0.4</td>
<td></td>
</tr>
<tr>
<td>W7</td>
<td>255/0.43</td>
<td>2187/0.76</td>
<td>2187/0.6</td>
<td></td>
</tr>
<tr>
<td>Mig</td>
<td>113628/13.6</td>
<td>22805/2.6</td>
<td>9185/1.7</td>
<td></td>
</tr>
<tr>
<td>Inv</td>
<td>over 620446/—</td>
<td>255781/39.2</td>
<td>135404/21.2</td>
<td></td>
</tr>
<tr>
<td>SC2</td>
<td>290/0.4</td>
<td>123/0.3</td>
<td>47/0.3</td>
<td></td>
</tr>
<tr>
<td>SC3</td>
<td>17741/4.6</td>
<td>2687/1.6</td>
<td>733/1.4</td>
<td></td>
</tr>
<tr>
<td>SC4</td>
<td>over 260928/—</td>
<td>59436/13.4</td>
<td>14173/11.9</td>
<td></td>
</tr>
<tr>
<td>Pftp</td>
<td>95241/11.0</td>
<td>187614/29.9</td>
<td>70653/19.2</td>
<td></td>
</tr>
<tr>
<td>Snoopy</td>
<td>16279/4.4</td>
<td>14305/2.7</td>
<td>8611/2.4</td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Number of states explored and time taken for reachability analysis by various algorithms on DSM protocols. “PV all” column is the result of running PV with selective caching disabled. “PV selective” column indicates the results of running PV with selective caching enabled. When SPIN couldn’t finish the search, time is shown as —.

clients. A client chooses a server and requests for a service. A service consists of a two round trip messages between server and client and some local computations. As can be seen, SPIN cannot handle 4 servers and 4 clients. The tool aborts search after generating 620,446 states.

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

5 Conclusions

We presented a new partial order reduction algorithm called Two phase and showed that it preserves LTL-X properties (Appendix A). By avoiding the proviso and using deterministic transitions to bring the reductions, the algorithm can bring better reductions than other algorithms on a number of practical protocols. Two phase algorithm is implemented in PV. Source code for PV is available in source code from [NG96]. Currently, we are planning to
verify the proofs presented in this paper using PVS [ORR+96]. We believe that verification
of Two phase would be much simpler than the verification of [Pel96] algorithm [CP95] for
two reasons. First, Two phase is intrinsically simpler than [Pel96] algorithm. Second, in
[CP95], the authors used HOL. Compared to HOL, PVS is much more mechanized, and
hence we expect the effort required to be substantially smaller. Verification effort using PVS
would be reported elsewhere.

References

[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 soft-
ware and hardware synchronization mechanisms for distributed shared memory
multiprocessors. Technical Report UUCS-96-011, University of Utah, Salt Lake
City, UT, USA, September 1996.

[CP95] Ching-Tsun Chou and Doron Peled. Formal verification of a partial-order re-
duction technique for model-checking. ftp://ftp.cs.ucla.edu/pub/chou/por.ps,


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

[God95] Patrice Godefroid. Partial-Order Methods for the Verification of Concurrent
Liege, 1994–95.


[HP96] Gerard J. Holzmann and Doron Peled. The state of SPIN. In Computer Aided

[NG96] Ratan Nalumasu and Ganesh Gopalakrishnan. The PV Tool. Univ. of Utah,

[NG97] Ratan Nalumasu and Ganesh Gopalakrishnan. A new partial order reduction
algorithm for concurrent system verification. In CHDL, Toledo, Spain, April


A Liveness

Lemma 1 The assert statement (line 24) in Two-phase holds.

Proof: To prove that the assert statement holds, we note that in every iteration, list contains top(stack), and list is added to V_r in every call to Two-phase. Thus, at the end of each call, top(stack) is in V_r. Since the recursive calls to Two-phase in the for each loop make each member of s to be the top of the stack, the assert statement holds.

Lemma 2 Let S = (q_1, q_2, ..., q_{i-1}, q_i, q_{i+1}, ..., q_m, \sigma) be a member of list that is added to V_r on line 17. If x = (q, \sigma) \xrightarrow{\delta}(q', \sigma') is a transition of P_i then at the time of termination of the algorithm, \exists \langle S_0, S_1, ..., S_t, S'_t \rangle \in E_r such that (a) S_0 = S, (b) Q_i(S_0) = Q_i(S_t) = ... = Q_i(S_t) = q, (c) \Sigma_i(S_0) = \Sigma_i(S_t) = ... = \Sigma_i(S_t) = \sigma, (d) Q_i(S') = q' and \Sigma(S') = \sigma' (i.e., S' is a P_i successor of S_i, S' is generated by the transition x from S_i).

In other words, every change in g and the state of a process P_i from (q, \sigma) to (q', \sigma') in the full state space G_f is reflected in the reduced state space G_r if S is added in line 17.

Proof: In phase I, list is constructed from top(stack) by examining only deterministic processes. Let r_0 = top(stack), and the order of the states added to list in the while loop of phase I be r_1, r_2, ..., r_m = s. In the then clause of phase II, all elements of list, viz. r_0, r_1, ..., r_m, are added to V_r. Let S = r_j for some j \in \{0, ..., m\}.

There are two cases to consider. (i) P_i is executed while generating the sequence \langle r_j, r_{j+1}, ..., r_m \rangle \in E_r (ii) P_i is not executed while generating the sequence \langle r_j, r_{j+1}, ..., r_m \rangle \in E_r in phase I. In the first case, since P_i is executed in phase I, we can conclude that P_i is deterministic in r_j. Let P_i be first executed at state r_k \in \{r_j, ..., r_{m-1}\} resulting in state r_{k+1}. In this case, the lemma holds with \langle S_0, S_1, ..., S_t, S'_t \rangle = \langle r_j, r_{j+1}, ..., r_k, r_{k+1} \rangle.

If P_i is not executed while generating the sequence \langle r_j, r_{j+1}, ..., r_m \rangle we will show that the lemma holds with \langle S_0, S_1, ..., S_t, S'_t \rangle = \langle r_j, r_{j+1}, ..., r_m, S'_t \rangle where S'_t is generated from r_m by x. Since P_i is not executed while generating r_{j+1}, ..., r_m, and g is not assigned in Phase I, we can conclude that T_j and T_m agree on the i^{th} component of the state (i.e., Q_i(r_j) = Q_i(r_m) = q_i), and on g (i.e., \Sigma(r_j) = \Sigma(r_m) = \sigma). Also note that s = r_m is fully expanded lines 18–23. Since s is fully expanded in phase II (lines 18–23) and (q, \sigma) \xrightarrow{\delta}(q', \sigma')
the lemma holds due to the \textbf{assert} statement on line 24 and the assignment to $E_r$ on line 25. Thus in this case the lemma holds with $\langle S_0, S_1, \ldots, S_t, S' \rangle = \langle r_j, r_{j+1}, \ldots, r_m, S' \rangle$.

\textbf{Lemma 3} Let $S = (q_1, q_2, \ldots, q_{i-1}, q_i, q_{i+1}, \ldots, q_n, \sigma)$ be a member of list that is added to $V_r$ on line 27. If $x = (q_i, \sigma) \xrightarrow{i} (q'_i, \sigma')$ is a transition of $P_i$ then at the time of termination of the algorithm, $\exists \langle S_0, S_1, \ldots, S_t, S' \rangle \in E_r$ such that (a) $S_0 = S$, (b) $Q_i(S_0) = Q_i(S_1) = \ldots = Q_i(S_t) = q_i$, (c) $\Sigma_i(S_0) = \Sigma_i(S_1) = \ldots = \Sigma_i(S_t) = \sigma$, (d) $Q_i(S') = q'_i$ and $\Sigma(S') = \sigma'$ (i.e., $S'$ is a $P_i$ successor of $S_t$, $S'$ is generated by the transition $x$ from $S_t$).

This is the counter part of Lemma 2 except that it deals with the states added in the outermost else clause.

\textbf{Proof}: The proof is based on the induction on the iteration at which the $S$ is added to $V_r$.

\textit{Induction basis}: During the first call to \texttt{Two-phase}, the outer \texttt{then} clause of phase II is executed, and hence the lemma holds vacuously.

\textit{Induction hypothesis}: Let the states added in the \texttt{else} clause during the first $l$ calls of \texttt{Two-phase} satisfy the lemma.

\textit{Induction step}: Consider the $l + 1$th call. If the \texttt{then} clause is executed in this call, the lemma holds vacuously. Otherwise, let list contain states \texttt{top(stack)} = $r_0, r_1, \ldots, r_m = s$ at the end of phase I of $l + 1$th call. Let $S = r_j$ for some $j \in \{0 \cdots m\}$.

There are two cases to consider. (i) $P_i$ is executed while generating the sequence $\langle r_j, r_{j+1}, \ldots, r_m \rangle \in E_r$ (ii) $P_i$ is not executed while generating the sequence $\langle r_j, r_{j+1}, \ldots, r_m \rangle \in E_r$ in phase I. In the first case, by reasoning as in the first case of Lemma 2, we can show that this lemma holds.

To prove in the second case, note that the global variable $g$ is not assigned in phase I, hence $\Sigma(r_m) = \Sigma(r_j = S) = \sigma$. Since, the \texttt{else} clause is executed, $s = r_m$ must have been added to $V_r$ either in the \texttt{then} clause of a previous call or in the \texttt{else} clause of a previous call. If \texttt{then} clause is executed, by Lemma 2 we can find a sequence $\langle t_0, t_1, \ldots, t_k, S' \rangle$ where $t_0 = r_m$. On the other hand, if else clause is executed, by induction hypothesis, we can find sequence $\langle t_0, t_1, \ldots, t_k, S' \rangle$ where $t_0 = r_m$. In both cases, the lemma holds with $\langle S_0, S_1, \ldots, S_t, S' \rangle = \langle r_j, r_{j+1}, \ldots, r_m, t_0, t_1, \ldots, S' \rangle$.

\textbf{Lemma 4} (Termination) \texttt{Two-phase} terminates after a finite number of calls.

\textbf{Proof}: There are two parts to the proof of termination: (a) eventually no new calls to \texttt{Two-phase} are made, and (b) the while loop in the phase I terminates. To prove (a), note that new calls to \texttt{Two-phase} are made only in the body of the outer "if" statement in the second phase. Before these calls are made, all elements of list are added to $V_r$. The precondition to execute the "if" statement is that $s$ is not in $V_r$. By construction of list, $s$ is in list. Thus the number of states in $V_r$ increases at least by one as a result of adding list to $V_r$. In other words, if number of states in $V_r$ before the $i^{th}$ level call of \texttt{Two-phase} is made is $k$, then the number of states in $V_r$ before $i + 1^{th}$ level call of \texttt{Two-phase} is made is at least $k + 1$. Thus the maximum depth of calls to \texttt{Two-phase} cannot exceed the number of states in the protocol, which is finite. To prove (b), note that one new state is added to list in each iteration of while loop. Again, since the number of states in the protocol is finite, eventually no new states can be added to list, thus the while loop terminates.
Theorem 1 (LTL-X) The graph constructed by Two-phase, \( G_r \), satisfies a LTL-X property \( P \) involving only the global variable \( g \) iff \( G_f \) satisfies \( P \).

Proof: (Informal) It is easy to see that \( G_r \) is a subgraph of \( G_f \). Hence, every trace of \( G_r \) is also a trace of \( G_f \). Now, we will show that for every trace in \( G_r \), there is a corresponding trace in \( G_f \). First, we observe that at the time of termination, initial state is in \( V_r \). From Lemmas 2 and 3 we can conclude that any sequence of changes in \( g \) that is present in \( G_f \) is also present in \( G_r \). Together, they imply that every trace of \( G_f \) is also a trace of \( G_r \).

A formal proof that for every trace of \( G_f \) there is a corresponding trace of \( G_r \) is a little more involved. Let \( T_f \) be a trace in \( G_f \), and \( T_r \) be a corresponding trace in \( G_r \). Some times, \( T_r \) can be constructed simply by reordering the commuting transitions of \( T_f \). In some cases, however, \( T_r \) might contain transitions that are not contained in \( T_f \) at all. For example, assume that state \( S \) appears in both \( T_f \) and \( T_r \), and that \( P_i \) is in a deterministic state in \( S \), and that \( P_i \) is not executed at all after \( S \) in \( T_f \). It is possible that \( P_i \) is executed at \( S \) in Phase I of Two phase. In this case all traces of \( G_r \) would contain the \( P_i \) transition. Thus, \( T_r \) contains a transition from \( P_i \) that is not executed in \( T_f \) at all. However, when the transitions are projected on to the global variable \( g \) alone, both traces would be equivalent (modulo stuttering). Thus we can conclude that Two phase preserves all LTL-X properties involving only \( g \).