A New Partial Order Reduction Algorithm for Concurrent System Verification

Ratan Nalumasu, Ganesh Gopalakrishnan

Department of Computer Science

University of Utah, Salt Lake City, UT 84112

{ratan,ganesh}@cs.utah.edu

Abstract

This paper presents a new partial order reduction algorithm called *Two phase* that is implemented in a verification tool, *PV* (Protocol Verifier). Two phase significantly reduces space and time requirements on many practically important protocols on which the partial order reduction algorithms implemented in previous tools (Godefroid 1995, Holzmann *et al.* 1994, Peled 1996) yield very little savings. This is primarily attributable to their use of a run-time *proviso* deciding which processes to run in a given state. Two phase avoids this proviso and follows a much simpler execution strategy that dramatically reduces the number of executions examined on a significant number of examples. We describe the Two phase algorithm, prove its correctness, and provide evidence of its superior performance on a number of examples including the directory based protocols of a multiprocessor under development.

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 (Clarke *et al.* 1986, Holzmann 1991, Dill 1996) have shown considerable promise in verification of real-world protocol verification problems and have been used with success on many industrial designs. Using most explicit state enumeration tools, a concurrent system is modeled as a set of concurrent processes communicating via shared variables (Dill 1996) and/or communication channels (Holzmann 1991) executing under an interleaving model. An important run-time optimization called partial-order

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.

```

process P
{  int l;  if      :: c ! 7 -> skip;
           :: d ? 1 -> skip;

    fi;
    l := 0; g := 1;
    if      :: l == 0 -> skip
           :: l != 0 -> skip
    fi }

```

Figure 1 A sample process to illustrate definitions

reductions (Peled 1996, Godefroid 1995, Valmari 1993) 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 (Avalanche), we observed that existing tools that support partial-order reductions (Godefroid 1995, Holzmann *et al.* 1994) failed to provide sufficient reductions. We traced this state explosion to their use of run-time *provisos* (explained later) 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 (Avalanche). 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 this paper, we prove that PV preserves all stutter-free safety properties. In (Nalumasu *et al.* 1996b), we also prove that *Two phase* preserves liveness properties in a limited setting. We also summarize experimental results on a number of examples.

2 BACKGROUND

The tools SPIN (Holzmann 1991) and PO-PACKAGE (Godefroid 1995) as well as our PV verifier use Promela (Holzmann 1991) as input language. In Promela, a concurrent program consists of a set of sequential processes communicating via a set of global variables and channels. Channels may have a capacity of zero or more. For zero capacity channels, the rendezvous communication discipline is employed. Any process that attempts to send a message on a full channel blocks until a message is removed from the channel. Similarly, any process attempting to receive a message from an empty channel blocks until a message becomes available on that channel. For the sake of simplicity we assume in this paper that a channel is a point to point connection between two processes with a non-zero capacity, *i.e.*, we do not consider the rendezvous communication. This allows us to focus here on the purely interleaved model.

We now present the above intuitions more formally, with the aid of Figure 1, where g is a global variable, l is a local variable, c is an output channel and d is an input channel for P , and guarded commands are written as *if ... fi*. Similar classifications are employed in other partial-order reduction related works. The state of a sequential process consists of a control state (“program counter”) and data state (the state of its local variables). In addition, the process can also access global variables and channels.

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

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

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

unconditionally safe: A *local* transition is said to be *unconditionally safe* if, for all states s , if the transition is *executable* (*non executable*) in s , then it remains *executable* (*non executable*) in state s' resulting from the execution of any sequence of transitions T by other processes from s . *unconditionally safe* is a static notion. Examples: $l:=0$, $l==0$, $l!=0$. In other words, if a transition (e.g. $l==0$ of P) is executable (*non executable*) in a state, then it remains executable (*non executable*) in any state attained after the execution of an arbitrary sequence of transitions of other processes from that state. As an illustration of something that is *not* unconditionally safe, consider the transition $d?l$. If channel d is empty in a state s , transition $d?l$ is not executable in s . However, if some other process executes transition $d!msg$ from s resulting in s' , transition $d?l$ becomes executable in s' . Thus, $d?l$ is *not* unconditionally safe. (However it is conditionally safe, as defined below).

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

after T . Another example of a conditionally safe transition is $d?1$, with the safe execution condition being that d be not empty.

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

deterministic: A process P is said to be *deterministic* in state s , written *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 executable. *deterministic* is a dynamic notion. Example: In Figure 1, if control state of P is at the second **if** statement, P is deterministic since only one of the two conditions $1==0$ and $1!=0$ can be true.

The Two phase algorithm performs the search in the following way. Whenever a state S is explored by the algorithm, in the first phase all deterministic processes are executed one after the other, resulting in a state S' . In the second phase, the algorithm explores *all* transitions executable at S' .

3 ALGORITHMS

The algorithm presented in (Holzmann *et al.* 1994) attempts to find a process in an internal state such that all transitions of that process from that internal state are safe and that none of the transitions of the process result in a state that is in the stack. This is also called strong proviso. If a process satisfying the above criterion can be found, then the algorithm examines all the executable transitions of that process. If no such process can be found, all executable transitions in that state are examined. This proviso preserves *all* LTL-X properties. To preserve safety properties only, the proviso can be weakened to guarantee that at least one of the transitions selected doesn't result in a state on the stack. This variation is called weak proviso.

In general, an algorithm using the strong proviso generates more states than another algorithm using the weak proviso, since the weak proviso can be satisfied more often than the strong proviso, and any time a process satisfying the above criterion cannot be found, all process in that state are examined by the algorithm. In this paper, we compare Two phase algorithm with an implementation that uses weak proviso (referred to as "Proviso algorithm" in rest of the paper). The proviso algorithm is shown in as *proviso()* in Figure 2. In this algorithm, *Choose*(s) is used to find a process satisfying the weak proviso. As mentioned earlier, the proviso (weak or strong) can cause the algorithm to generate many unnecessary states. In some protocols, e.g., Figure 3 (a), all reachable states in the protocol are generated. Figure 3 (c) shows the state space generated on this protocol. Another algorithm that uses the (weak) proviso and sleepsets implemented in the tool PO-PACKAGE, (Godefroid 1995), also exhibits similar state explosion.

The Two phase algorithm is shown as *Twophase()* in Figure 2. In the first

<pre> init stack to contain initial state init cache to contain initial state proviso() { s := top(stack); (i, found) := Choose (s); if (found) { tr := {t t is executable in s and PID(t)=i}; nxt := successors of s obtained by executing transitions in tr; } else { tr := all executable transitions from s; nxt := successors of s obtained by executing transitions in tr; } for each succ in nxt do { if succ not in cache then cache := cache + {succ}; push(succ, stack); proviso(); } } pop(stack); } </pre>	<pre> init stack to contain initial state init cache to Φ Twophase() { 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 Pi */ s := next(i, s); if (s $\in$ list) goto NEXT_PROC; list := list + {s}; } NEXT_PROC: /* next i */ } /* Phase II: classical DFS */ if (s $\notin$ cache) { cache := cache + list; nxt := all successors of s; for each succ in nxt { if(succ$\notin$cache) push(succ, stack); Twophase(); } } else { cache := cache + list; } pop (stack); } </pre>
---	---

Figure 2: proviso() is a partial order reduction algorithm using weak proviso. Twophase() avoids proviso using a different execution strategy.

phase, *Twophase()* executes deterministic processes. States generated in this phase are saved in the temporary variable *list*. These states are added to *cache* during the second phase. In the second phase, *all* executable transitions at *s* are examined.

The Two phase algorithm outperforms the proviso algorithm and PO-PACKAGE (Godefroid 1995) when the proviso is invoked often; confirmed by the examples in Section 5. In most reactive systems, a transaction typically involves a subset of processes. For example, in a server-client model of computation, a server and a client may communicate without any interruption from other servers or clients to complete a transaction. After the transaction is completed, the state of the system is reset to the initial state. If the partial order reduction algorithm uses the proviso, state resetting cannot be done as the initial state will be in the stack until the entire reachability analysis is completed. Since at least one process is not reset, the algorithm generates

unnecessary states, thus increasing the number of states visited. As shown in Figure 3, in certain examples, *proviso()* generates all the reachable configurations of the systems. In realistic systems also the number of extra states generated due to the proviso can be high. Two phase does not use the proviso, thus avoiding generating the extra states.

4 CORRECTNESS OF THE TWO PHASE ALGORITHM

We show that *Twophase()* preserves all stutter free safety properties. To establish the correctness of *Twophase()*, we need the following two lemmas.

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

Proof: Based on induction on the “time” a state is entered into **cache**.

Induction Basis: During the first call of *Twophase()* the outer “if” statement of the second phase will be executed; during this phase, all states in **list** are added to **cache** in the body of the “if” statement. Following that the algorithm examines all successors of s . Let s' be an arbitrary element of **list**. By the manner in which **list** is generated, s' can reach s via zero or more deterministic transitions. By the definition of deterministic transition, any executable transition at s' , but not executed in any of the states along the path from s' to s will remain executable at s . Since all transitions out of s are considered in the second phase, it follows that all unexecuted transitions out of s' are also considered. Hence the addition of s' to **cache** satisfies Lemma 1.

Induction Hypothesis: Let the states entered into **cache** during the first $i - 1$ calls to *Twophase()* be $s_1, s_2, \dots, s_n$. Assume by induction hypothesis that all executable transitions at every state s_i in this list are guaranteed to be executed at s_i or a successor of s_i .

Induction Step: We wish to establish that the states entered into **cache** during the i^{th} call to *Twophase()* also satisfy the Lemma. There are two cases to consider: (1) the outer “if” statement of the second phase is executed; (2) the “else” statement of the second phase is executed. In the first case, an argument similar to the one used in induction basis can establish that the lemma is not violated. In the second case, s is already in **cache**; it was entered during an earlier call to *Twophase()*. By induction hypothesis, all executable transitions of s are already executed or guaranteed to be executed. Let s' be an arbitrary member of **list**. Hence all executable transitions of s' were either already considered at s or guaranteed to be executed by the hypothesis. Hence adding s' to **cache** does not violate the lemma. Thus in both cases, **list** can be added to **cache** without violating the lemma.

Lemma 2: *Twophase()* terminates after a finite number of calls.

Proof: Omitted. See (Nalumasu *et al.* 1996) for details.

Theorem: *Twophase()* preserves all stutter free safety properties.

N	Proviso	PO-PACKAGE	Two phase
B5	243/0.34	217/0.42	11/0.33
B6	729/0.38	683/0.64	13/0.33
B7	2187/0.50	2113/1.4	15/0.33
W5	63/0.33	64/0.37	243/0.39
W6	127/0.39	128/0.42	729/0.49
W7	255/0.43	256/0.51	2187/0.76

Table 1 Number of states explored and time taken for reachability analysis by various algorithms on the Best Case and the Worst Case.

Proof: (*Informal*) The proof of the theorem follows from the observation that establishing stutter free safety properties about a finite-state model requires every executable transition be executed. Further, a transition need not be executed at a state if it is executed at a successor of that state obtained by executing a sequence of safe transitions (*i.e.* stuttering steps do not matter). From Lemma 1, and the definition of *deterministic*, *Twophase()* satisfies these two conditions. See (Nalumasu *et al.* 1996) for details.

5 CASE STUDIES

Figures 3 and 4 show two extremal cases. Table 1 shows the results of running the algorithms on these two protocols. In this table rows B5–B7 show the results of running the best case protocol (Figure 3) and W5–W7 show the results of running the worst case protocol (Figure 4). For the best case protocol with n processes, Two phase generates $2n + 1$ states whereas the proviso algorithm generates 3^n states. The reason for better performance of Two phase is that the initial state is reached many times during the DFS analysis by the proviso algorithm. Since the initial state is always in the stack, the proviso is invoked many times, thus increasing the number of states visited by the algorithm. On the other hand, for the worst case protocol with n processes, the Two phase algorithm generates 3^n states whereas the proviso algorithm generates $2^{(n+1)} - 1$ states. The reason for the poor performance of Two phase algorithm on this protocol is that none of the reachable states is deterministic with respect to any process. Hence, Two phase degenerates to classical depth first search.

Several realistic directory-based distributed shared memory protocols from the Avalanche multiprocessor project (Avalanche) underway at the University of Utah were experimented with. Some of the well known directory based coherency protocols are *write invalidate*, *write update*, and *migratory*. Table 2 shows the results of running the different algorithms on the migratory and the invalidate protocols. The last two rows of the table show the results when the

Figure 3: Best case protocol. (a) The protocol. (b) State space that generated by the 2-phase algorithms. (c) State space generated by SPIN using the weak proviso. Dotted lines in (c) show some of the transitions that were not attempted due to the proviso. The thick line in (c) shows one of the transitions that will be taken by the algorithm, but find that the state is already generated.

Figure 4: Worst case protocol.

TwoPhase() is modified as below. In Phase-I of the algorithm all intermediate states are added to **list**, and later **list** is added to **cache**. The purpose of adding the states to **list** is to ensure that the **while** loop in the Phase-I terminates. Instead of adding **s** to **list** in every iteration of the **while**, we can add **s** only when **next(i, s)** is bitwise smaller than or equal to **s**. This modification, referred to as selective caching, also guarantees that the loop terminates. This modification results in a substantial improvement in the performance. The results with these modifications are shown as the last two rows in Table 2. All the verification runs were limited to 64 MB of memory.

Protocol	Proviso	PO-PACKAGE	Two phase
Migratory	34906/5.08	28826/14.45	23163/2.84
Invalidate	Unfinished	Unfinished	193389/19.23
Migratory			9185//1.10
Invalidate			70004/6.20

Table 2 Number of states explored and time taken for reachability analysis by various algorithms on DSM protocols. The last two rows indicate the results of running Twophase() with selective caching.

Protocol	Proviso	PO-PACKAGE	Two phase
W6	281/0.53		172/0.42
W7	384/0.68		230/0.47
W8	503/0.91		296/0.56
Sort	174/0.35	173/0.6	174/0.33
Snoopy	20323/6.22	10311/10.53	20186/5.08
Pftp	161751/34.5	125877/150.7	230862/36.3
SC3	11186/3.43	8639/7.74	3232/0.83
SC4	Unfinished	Unfinished	62025/14.9

Table 3 Number of states explored and time taken for reachability analysis by various algorithms on wave front arbiter, the protocols supplied as part of SPIN distribution, and the server/client protocol.

The proviso algorithm aborted search on *invalidate* after generating more than 963,000 states due to lack of memory.

A cross-bar arbiter that operates by sweeping diagonally propagating “wave-fronts” within a circuit array (Gopalakrishnan 1994) was verified using PV. The results are presented as rows W6–W8 in table 3. Statistics for PO-PACKAGE algorithm is not reported on this example as the protocol contains a large number of processes that the implementation could not handle.

We also ran Two phase on the protocols provided as part of the SPIN distribution. Some of the protocols supplied with the SPIN distribution are not perpetual processes (i.e., they terminate or deadlock). *Sort* protocol sorts a sequence of numbers. This protocol has no non-determinism and terminates after a finite number of steps, hence the proviso algorithm and Two phase generate equal number of states. *Snoopy* is a cache coherency protocol to maintain

consistency in a bus based multiprocessor system. This protocol contains a large number of deadlocks, and therefore Two phase is not as effective. *Pftp* is a flow control protocol. This protocol contains little determinacy. Hence Two phase algorithm is not as effective. *SC3* is a server client protocols we designed that contains 3 clients that communicate with one of the 3 servers at a time to complete a transaction. *SC4* is a similar protocol except that it contains 4 clients and 4 servers. On this protocol, the proviso is invoked each time after a transaction is completed. Hence, the Two phase algorithm outperforms the other two algorithms. Run times of these protocols are summarized in Table 3.

6 CONCLUSIONS

We have presented a new partial order reduction algorithm called Two phase. Unlike the proviso algorithm or PO-PACKAGE (Godefroid 1995), Two phase does not use the proviso. Instead it alternates one step of partial order reduction step (using deterministic transitions) followed by one step of classical depth first search (using all transitions). The (strong) proviso algorithm is shown to preserve all LTL-X formulae (Peled 1996). In this paper, we proved that Two phase algorithm preserves only stutter free safety properties. It is not difficult to see that Two phase also preserves liveness properties. (Nalumasu *et al.* 1996b) presents a proof that Two phase preserves liveness properties in a limited setting where all transitions are either unconditionally safe or unsafe, i.e., conditionally safe transitions are not present.

REFERENCES

- Avalanche. <http://www.cs.utah.edu/projects/avalanche>.
- Clarke, E. M., Emerson, E. A., and Sistla, A. P. (1986). Automatic verification of finite-state concurrent systems. *ACM TOPLAS*, 8(2):244–263.
- Dill, D. (1996). The Stanford Murphi verifier. In *CAV*, New Brunswick, New Jersey.
- Godefroid, P. (1995). *Partial-Order Methods for the Verification of Concurrent Systems: An approach to the State-Explosion Problem*. PhD thesis, Univerite De Liege.
- Gopalakrishnan, G. (1994). Micropipeline wavefront arbiters. *IEEE Design & Test of Computers*, 11(4):55–64.
- Holzmann, G. (1991). *Design and Validation of Computer Protocols*. Prentice Hall.
- Holzmann, G. and Peled, D. (1994). An improvement in formal verification. In *FORTE*, Bern, Switzerland.
- Nalumasu, R. and Gopalakrishnan, G. (1996). Partial order reductions without the proviso. Technical Report UUCS-96-006, University of Utah, Salt Lake City, UT, USA.
- Nalumasu, R. and Gopalakrishnan, G. (1996b). Liveness proof of the two-phase algorithm. <http://www.cs.utah.edu/~ratan/twolive.html>.
- Peled, D. (1996). Combining partial order reductions with on-the-fly model-checking. *Journal of Formal Methods in Systems Design*, 8 (1):39–64.
- Valmari, A. (1993). On-the-fly verification with stubborn sets. In *CAV*, pages 397–408, Elounda, Greece.