Saving Space by Fully Exploiting Invisible Transitions

Hillel Miller[*] and Shmuel Katz[**]

Department of Computer Science
The Technion
Haifa, Israel
hillelm,katz@cs.technion.ac.il

Abstract. Checking that a given finite state program satisfies a linear temporal logic property suffers from a severe space and time explosion. One way to cope with this is to reduce the state graph used for model checking. We present an algorithm for constructing a state graph that is a projection of the program's state graph. The algorithm maintains the transitions and states that affect the truth of the property to be checked. The algorithm works in conjunction with a partial order reduction algorithm. We show a substantial reduction in memory over current partial order reduction methods, both in the precomputation stage, and in the result presented to a model checker, with a price of a single additional traversal of the graph obtained with partial order reduction. As part of our space-saving methods, we present a new way to exploit Holzmann's Bit Hash Table, which assists us in solving the revisiting problem.

1 Introduction

In order to reduce the space needed for model checking of linear temporal logic properties, several pre-processing techniques have been suggested to construct smaller graphs such that a property to be checked is true of the original state graph iff it is true of the reduced graph.

In particular, partial order methods such as [GW91],[Val90], and [Pel94] exploit the fact that certain operations are independent of other operations, and that not all interleavings of independent operations need to be explicitly examined. Here, we exploit the fact that the specification –the property to be proven– is independent of certain operations to obtain a further reduction. Invisible operations are those that do not affect the truth of any of the atomic propositions of the specification, while visible operations do affect them. A node is considered visible if some edge corresponding to a visible operation enters it, and invisible otherwise. We also exploit the fact that an operation can be invisible or visible, depending on the state from which it is executed. In this paper, a program's projected visible state space relative to a specification is constructed through a

[*] Presently with Motorola Semiconductors–Israel, Herzlia, Israel
[**] Supported by the Technion V.P.R. Fund–Promotion of Sponsored Research

DFS traversal, and the invisible states are eliminated. Thus we present to the model checker a much smaller structure that represents the program.

The construction of the visible state space requires a linear traversal of a state graph that is somewhat reduced from the original, but can still be large in some cases. This is still worthwhile because a standard temporal logic model checker requires space and time complexity which is the multiplication of the size of the state space by a term exponential in the length of the formula. Thus for a formula of length 20, the time and memory complexity for the model checker are multiplied by 10^6. We are therefore motivated to reduce the state graph given to the model checker.

Moreover, we will show that the reduced structure can be produced with a low space overhead. During any such pre-processing, and also during a traversal of the state-space for purposes of reachability analysis and deadlock detection, the question arises of whether to record for future reference that a particular state was already visited. Not indicating that a state was visited saves space, but may be costly in time: if the state is later reached again along another path, its descendants must be recomputed unnecessarily. We can define the revisiting degree of a state as the number of incoming edges not including those that close loops. (Those that close loops are on the stack used for a depth-first traversal, and thus are easily identified with no additional space needed). The question is whether a state should be identifiable as having already been visited even after backtracking from that state, in the DFS traversal. For graphs with many states having a large revisiting degree, the time can increase exponentially if states are reexpanded each time they are reached. Identifying such states avoids this problem, but can lead to memory overflow. This trade-off can be called the state revisiting problem.

In [GHP92], the state revisiting problem is considered, for reachability analysis, in the context of a partial order reduction method. Their conclusion is that in that context, the state revisiting problem can be ignored, states should not be saved after visiting, and that the price to be paid in recomputation is tolerable (3 to 4 times a single traversal, for their examples). In general model checking, however, the number of recomputations can be unacceptably large.

It is clear that partial order reductions as in [GHP92] lessen the state revisiting problem because one cause of reaching the same state by different paths is that independent operations are executed in a different order. Nevertheless, recomputation is still sometimes necessary both because such methods only eliminate some of the redundancy of various orderings of independent operations, and because sometimes the same state is reached through truly different sequences of operations. Partial order methods consider operations dependent and/or influencing the specification, if they *might* have such an influence. Here we check more carefully whether an occurrence of an operation actually affects atomic clauses in the specification, as in [KP92], and thus can have greater savings.

Another approach to the state revisiting problem was proposed in [Hol88] by using a hash table where the keys are the states themselves, to indicate whether a state has already been visited, without saving the full states. The difficulty with this approach is that there may be a hash conflict, and then a state can map to a hash entry indicating that it has been visited, even if it has not been, and thus some parts of the graph may never be explored. Here we both achieve a greater reduction in the graph to be used for model checking, and overcome the state revisiting problem while exploring all of the state space, at a cost of a single additional traversal of the graph obtained by the partial order expansion, beyond the one needed by the partial order method itself.

In order to overcome the state recomputation problem, a preliminary DFS traversal is used to compute the revisiting degree of each state, so that it is clear, on the second traversal, when a state should be retained, and when it can be eliminated. Thus, we can manage a caching method that does not randomly free memory. During this preliminary traversal, a hashing method similar to that in [Hol88] can be used. A significant difference is that when conflicts do occur in the hash table, the worst effect will be some additional recomputation, but the entire graph will ultimately be examined.

In the following section some preliminary definitions are given, the visible state graph is defined and theorems with its properties are given. In Section 3 the basic traversal algorithm to eliminate invisible states is first described, then related to a partial order method, and finally combined with a preliminary traversal to determine revisiting degrees. In Section 4 we summarize the memory and time complexity both of the pre-computation, and of the graph presented to the model checker, and present some simulation results.

In Figures 1 to 3, an example program, and several of its state-space graphs are shown. The specification is of mutual exclusion and of liveness. Y_1 and Y_2 represent flags that are true when processes P_1 and P_2 respectively are in crucial sections. The assertion is that Y_1 and Y_2 are never both true at the same time, and that if one flag is true, the other will eventually become true. The program is represented as a labeled set of guarded commands, for each of the two processes. Each process has its own program counter (denoted PC_i), to control the internal flow of the process. A command is enabled if its guard is true and if the program counter of the process containing it is equal to the command's number. Figure 2 shows the full state-space graph, while the graph without the grey nodes and the dotted lines is the reduced graph after the partial order method of [Pel94] is applied. In Figure 3 the graph is shown in an intermediate stage, after some of the invisible states have been removed, with the candidates for elimination in the rest of the algorithm indicated in gray. The graph without the grey nodes, and with edges connected to their successors is the fully reduced graph relative to the given specification. Note that the original graph has 30 states, the one after partial order reduction has 26, and the graph that fully exploits the elimination of invisible states has only 14. The stages in this reduction will be explained later in the paper.

In this toy example, the specification includes both of the program variables, and only operations involving the control counters are invisible. When the program is more realistic, and the property to be proven only involves part of the variables, much greater savings can be expected, as is shown in the simulations summarized in Section 4.

This paper demonstrates that a careful combination of a partial order reduction method with an algorithm to eliminate states not relevant for the specification, along with a hashing technique to save only relevant information about which states have already been considered, can yield a result that makes previously infeasible problems treatable.

- Global State Representation = (PC_1, PC_2, Y_1, Y_2)
- Initial State = $(0, 0, F, F)$
- Specification Checked = $\Box\neg((Y_1 = T) \wedge (Y_2 = T)), \Box((Y_1 = T) \Rightarrow \Diamond(Y_2 = T))$

PROCESS 1	PROCESS 2
PC1	PC2
0: PC_1:=1 **{a1}**	0: PC_2:=1 **{b1}**
1: PC_1:=2; Y_2:=T **{a2}**	1: PC_2:=2; Y_1:=T **{b2}**
2: (Y_1=T) $\Rightarrow$ PC_1:=3; Y_1:=F **{a3}**	2: (Y_2=T) $\Rightarrow$ PC_2:=3;Y_2:=F **{b3}**
3: PC_1:=0 **{a4}**	3: PC_2:=0 **{b4}**

Fig. 1. Example of a program P.

2 Preliminaries

A finite state program P is a triple $< T, Q, I >$ where T is a finite set of operations, Q is a finite set of states, and $I \in Q$ is the initial state. The enabling condition $en_\alpha \subseteq Q$ of an operation $\alpha \in T$ is the set of states from which α can be executed. Each operation $\alpha \in T$ is a partial transformation $\alpha : Q \mapsto Q$ which needs to be defined at least for each $q \in en_\alpha$. For simplicity we assume that for each $q \in Q$ there exists an operation $\alpha \in T$ such that $q \in en_\alpha$.

An interleaving sequence of a program is an infinite sequence of operations $v = \alpha_0\alpha_1\ldots$ that generates the sequence of states $\zeta = q_0q_1q_2\ldots$ from Q such that (1) $q_0 = I$, (2) for each $0 \leq i, q_i \in en_{\alpha_i}$ and $q_{i+1} = \alpha_i(q_i)$.

A nexttime-free LTL formula (denoted LTL-X) is composed of atomic propositions from a set AP, boolean operators $(\wedge, \neg, \vee)$ and the usual temporal modals $\Box$ ('always'), $\Diamond$ ('eventually') and U ('Until') but not the modal $\bigcirc$ ('next').

Definition 1. A *state graph*, $G_P = (\hat{s}, S, E)$, for a program P is a directed, rooted graph, such that :

1. S is a finite set of nodes, $\hat{s} \in S$ is the graph's root and E is a finite set of edges (we denote an edge from node s to node t as $s \longrightarrow t$).
2. The graph is total, i.e. from every node there is an exiting edge.

3. There is an injective homomorphism $st : S \longrightarrow Q$ that maps nodes to program states such that: $st(\hat{s}) = I$ and if $s \longrightarrow t \in E$ then there exists an operation α such that $st(s) \in en_\alpha$ and $st(t) = \alpha(st(s))$.
4. The graph is maximal, i.e, for each state s in a sequence of states generated from some sequence of operations from P, and for each operation α enabled at s such that $\alpha(s) = t$, we have that $st^{-1}(s) \longrightarrow st^{-1}(t) \in E$.

We will identify a state and a node with this mapping.

Definition 2. $M = (G_P, V)$ is a *model for a program P and a specification φ* iff G_P is a state graph of P and V is a function $V : S \longrightarrow 2^{AP}$ (where AP are the atomic propositions of φ) such that for all nodes $s \in S$, $V(s) = \{a \mid a \in AP$ and a is true in state $st(s)\}$.

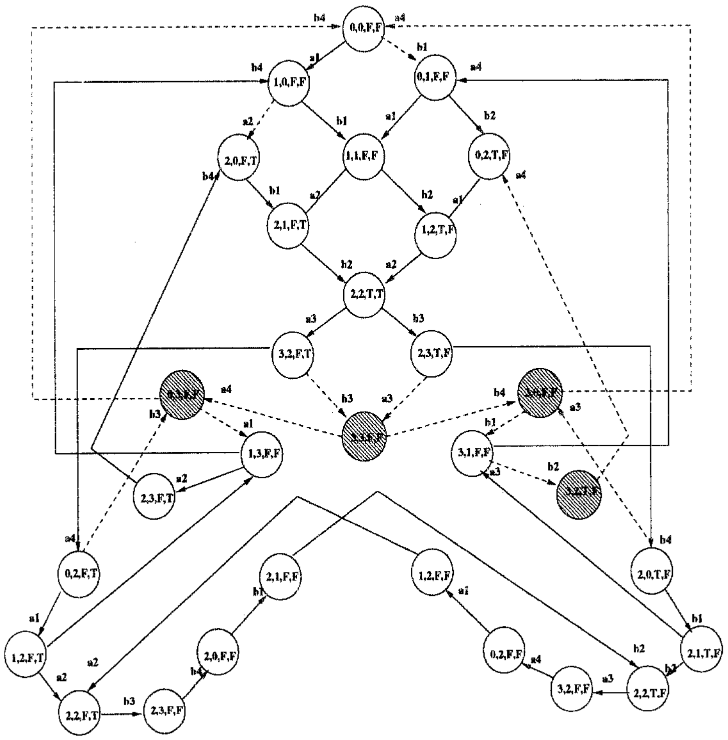

Fig. 2. P's full and partial ordered state graph.

Definition 3. Let $M(G_P(\hat{s}, S, E), V)$ be a model of a program P and specification φ, $s \longrightarrow s' \in E$ is a *visible edge* iff $V(s) \neq V(s')$. A node t is a *visible node* iff 1) $t = \hat{s}$ (i.e. the initial node) or, 2) there is a visible edge entering t.

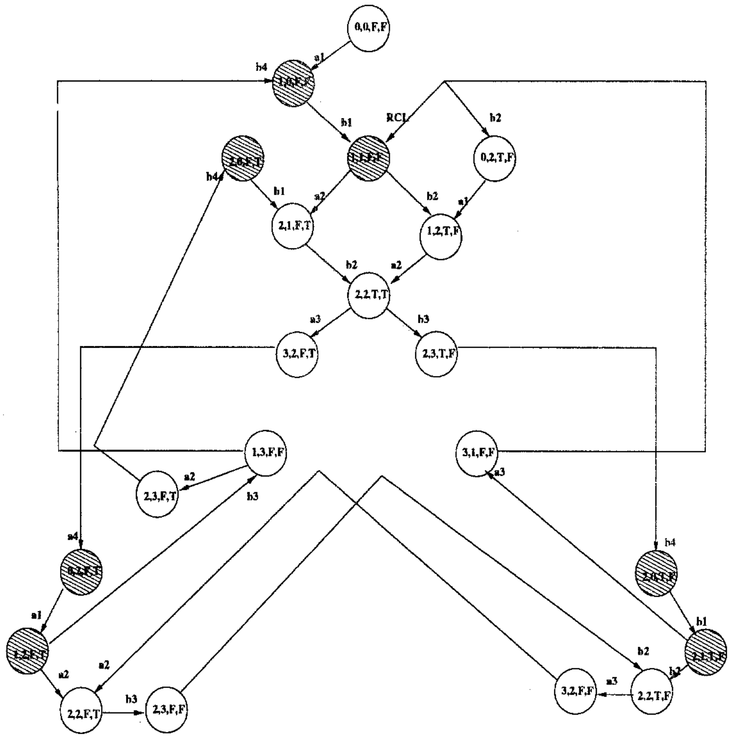

Fig. 3. An intermediate stage in the reduction algorithm.

The visible state graph G_V is the state graph of an abstraction of a program P. Its set of nodes is exactly the visible nodes of the state graph G_P (denoted $VIS(G_P)$). Its edges satisfy the following two properties:

P1 For any 2 nodes $s, s' \in \text{VIS}(G_P)$, there is a sub-path $ss_1 \ldots s_n s'$ of a path of G_P such that $V(s) = V(s_1) = \ldots = V(s_n)$ and $V(s) \neq V(s')$ iff there is subpath $st_1t_2 \ldots t_m s'$ of a path of G_V such that $V(s) = V(t_1) = \ldots = V(t_m)$ and $V(s) \neq V(s')$

P2 For any node $s \in VIS(G_P)$, there is a suffix $ss_1s_2 \ldots$ of a path of G_P such that $V(s) = V(s_1) = V(s_2) = \ldots$ iff there is a suffix $st_1t_2 \ldots$ of a path of G_V such that $V(s) = V(t_1) = V(t_2) = \ldots$.

These properties guarantee that paths in the two graphs have the same properties when repetitions of the relevant truth values are ignored, i.e., they are *stuttering equivalent* [LAM83] (denoted by $\pi \sim \pi'$).

Theorem 4. *Let G_P and G_V be the state graph and a visible state graph respectively of a program P. For each path π in G_P there exists a path π' in G_V such that $\pi \sim \pi'$. For each path π in G_V there exists a path π in G_P such that $\pi \sim \pi'$.*

The proof is done by building a linear stuttering equivalent relation based on properties P1 and P2, and using the fact that LTL-X is insensitive to stuttering.

3 Visible state graph generation

In this section we first show the basic algorithm (Figure 4) which generates the visible state graph. In the algorithm we do not keep an indicator whether invisible nodes have been visited (after backtracking from it). Therefore the time may increase exponentially for graphs that have many invisible nodes with high revisiting degrees. Later we show how to solve this problem.

In the algorithm we abstract from implementation details. We create a visible state graph G, which is represented by a set of nodes S and a set of edges E. We operate on the sets with the standard set operations $(\cup, \cap, \setminus)$ and operands $(\in, \subseteq)$. The search path is kept on a stack. In intermediate stages, S will contain both the visible nodes already examined and all nodes on the stack. The algorithm uses the following functions and indicators:

- $s \in S$ - Either s is a visible node already examined or is on the search stack.
- $s \longrightarrow s' \in E$ - An edge from node s to visible node s' was created.
- **en(s)** - The set of operations enabled at state $st(s)$.
- $\alpha(s)$ - The node obtained after executing an operation α on state $st(s)$.
- **visible(s)** - The node s is visible (only for nodes in S).
- **RCL**$(s \longrightarrow s')$ - The edge $s \longrightarrow s'$ must be reconnected when backtracking from s'.
- **open(s)** - Node s is on the search stack.

The algorithm is based on a standard DFS traversal, implemented in a recursive procedure: At each node s we calculate its set of successors. We then recursively examine all successors that are not indicated as having already been examined (remember we sometimes reexamine a node more than once). The reduction comes when backtracking from a successor s' of the node s. If s' is invisible we replace each edge exiting s' with an edge that exits s and that enters the same target. We then remove the set of edges exiting s', which is followed by the removal of the invisible node s' (lines 10 - 16). In lines 13-14 before removing an edge that is marked RCL (see explanation below) and exiting s' we mark the respective replacing edge that exits s as RCL. In line 11 if s' has a self loop we give s a self loop. This maintains diverging sequences. Note that even if the state later proves to be visible when approached along a different path, and is therefore reintroduced, the edges we remove are invisible. When s' is visible we add the edge $s \longrightarrow s'$ to E (line 18).

If a successor s' of s is in S then this indicates that either s' is visible and has been examined or s' is open (i.e. s' closes a loop), thus we add the edge $s \longrightarrow s'$ to E. If s' closes a loop and $s \longrightarrow s'$ is invisible (i.e $V(s) = V(s')$) we mark that edge RCL, standing for *reconnect later*. In the DFS traversal when we arrive at an invisible open node s' from s, there may be successors of s' that have not yet been examined. We therefore do not know all the visible successors of s' and we cannot know which are s's successors (that go through s' in the original graph) in the visible graph. Hence, we indicate (i.e. RCL$(s \rightarrow s')$:= TRUE in line 24) that we still have to update the set of edges exiting s. Finally when we backtrack from s', the sub-tree from s' has been examined. Thus we know which are s''s visible successors. We then replace the set of edges which enter s' and that are marked RCL (lines 25-28) by edges that enter s''s visible successors. Note that any edge entering s' that is marked RCL when we backtrack from s' is from a visible node or a self loop from s' (because an edge marked RCL closed a loop, the node it came from has already been backtracked from, and was removed if it was invisible).

```
1    procedure expand(s)
2       open(s) := TRUE
3       foreach $\alpha \in$ en(s) do
4          $s' := \alpha(s)$
5          if not $(s' \in S)$ then
6             visible($s'$):=FALSE
7             $S := S \cup \{s'\}$
8             expand($s'$)
9             if (not visible($s'$)) and $(V(s) = V(s'))$ then
10               foreach $u$ such that $(s' \rightarrow u) \in E$
11                  if $s' = u$ then $u := s$
12                  $E := E \cup \{s \rightarrow u\}$
13                  if RCL$(s' \rightarrow u) and (s' \neq u)$ then
14                     RCL$(s \rightarrow u)$ := TRUE ;RCL$(s' \rightarrow u)$ := FALSE ;
15                  $E := E \setminus \{s' \rightarrow u\}$
16               $S := S \setminus \{s'\}$
17            else
18               $E := E \cup \{s \rightarrow s'\}$
19               visible($s'$) := TRUE
20         else
21            if $V(s) \neq V(s')$ then visible($s'$) := $TRUE$
22            $E := E \cup \{s \rightarrow s'\}$
23            if open($s'$) and $(V(s) = V(s'))$ then
24               RCL$(s \rightarrow s')$ := TRUE
25      foreach $u$ such that RCL$(u \rightarrow s)$
26         foreach $v$ such that $(s \rightarrow v) \in E$
27            $E := E \cup \{u \rightarrow v\}$
28            if RCL$(s \rightarrow v)$ then RCL$(u \rightarrow v)$ := TRUE
29         RCL$(u \rightarrow s)$ := FALSE
30      open(s) := FALSE
31   end
```

Fig. 4. Algorithm for generating the visible state graph.

We demonstrate the backtracking in Figure 2 where nodes are represented by a 4-tuple of values (PC1,PC2,Y1,Y2). In Figure 2 the algorithm first starts backtracking when it does a step from (2,1,F,F) and arrives a second time at node (2,2,T,F) (on the lower right). This node is visible because it has a visible operation (b2) entering it (e.g., the truth value of the atomic proposition "Y1=T" is changed). Therefore it is not deleted when backtracking from it. On the other hand, node (2,1,F,F) has only an invisible operation (b1) entering it (because only the program counter, irrelevant to the specification, is changed). Therefore it is deleted when backtracking. Its successors are now added to the successor set of its predecessor (state (2,0,F,F)). Next, node (2,0,F,F) is also deleted because it is an invisible node. Its successor (i.e. node (2,2,T,F)) will now become a successor of the visible node (2,3,F,F). The result of the above can be seen in Figure 3.

The use of RCL can be seen in Figure 3 where invisible node (1,1,F,F) (the second from the top center) has an edge marked RCL entering it. When backtracking from node (1,1,F,F), before removing it we reconnect node (3,1,F,F) to the nodes (2,1,F,T), and (1,2,T,F).

To show the algorithm correct we must prove that properties P1 and P2 hold with respect to the full state graph and the graph constructed (i.e, it is a visible state graph). This is done by induction on the set of backtracking steps executed by the algorithm. For each step of the induction we look at: (1) the (intermediate) full state graph obtained from the edges backtracked from, (2) the (intermediate) graph obtained from edges in the set E (see algorithm). We then show that an intermediate version of P1 and P2 hold for these two graphs. When the algorithm terminates, the "full" version of P1 and P2 hold.

We can combine our algorithm with any algorithm for partial order reduction, e.g., A1 from [Pel94]. In that algorithm we execute a DFS traversal of a program's state space. At each state only a subset of the enabled transitions (called the ample set) are expanded. This is due to the fact that expanding all enabled transitions will lead to a graph with more than one interleaving per partial order. The only change to our algorithm is that instead of expanding all the enabled operations from a particular state s, we expand only those operations that belong to the ample set of state s. Therefore, we replace line 3 in Figure 4 with: **foreach** $\alpha \in$ **ample(s) do**. Other algorithms differ in the way that a subset of enabled transitions are selected, but can be used in the same way.

To solve the revisiting problem, we present an algorithm that pre–processes the state space. The algorithm calculates the revisiting degree of each state. This information is passed on to the algorithm that generates the visible state graph, which then can more selectively delete states. The preliminary DFS algorithm traverses the state space in a partial order manner, which is the exact same order used in the later reduction algorithm (both are deterministic).

We use a Hash Table [Hol88] (called the *revisited* hash table), as a revisiting degree counter for each node. The hash table is accessed with a hash function whose argument is a state. When visiting a node in the DFS traversal, we check if its revisiting degree is zero. If this is the case we set its revisiting degree counter to 1, and then we recursively calculate the revisiting degree of all its successors (from the *ample* set of the underlying partial order traversal). Otherwise, we increase the counter of the node by 1, and backtrack from that node. Note that all this only relates to revisits that do not close a loop, for reasons explained already in the Introduction.

Here we use Holzmann's hash table to assist us in calculating the revisiting degree of each node. This is a novelty in itself: until now the use of this technique was problematic, because of the small probability of a hash collision when model checking (resulting in not checking part of the state space). Here in the worst case, a hash collision will cause us to calculate an incorrect revisiting degree, resulting in additional state recomputation in the latter DFS.

Now, in the latter DFS that generates the visible state graph, when backtracking from a state, we check if the node will be revisited (according to the *revisited* hash table). If an invisible node will not be revisited we remove it and all its pointers to its successors from internal memory, otherwise we maintain it and pointers to its successors in memory. (If we were in error because of hash conflicts, we might have to regenerate the node and its pointers later on, when we reach that state along another path.) If a visible node will not be revisited we remove it from the internal memory and store it on external memory, otherwise we maintain it in memory. The pointers of a visible node to its successors are always stored in external memory, because they are not needed for later revisits in the traversal. When we revisit such a node we decrement its counter in the *revisited* hash table.

When we run out of memory, we can do a form of garbage collection: For each node s in memory (stored in a balanced tree) we check if s's counter is zero. If this is the case, we delete that state from the internal memory, and store it on the external memory (if it is visible). Note that some nodes may have the same entry in the *revisited* hash table. This means that they can only be deleted together (i.e. when they all will not be revisited anymore). Thus we must execute a garbage collection to dispose of these kinds of nodes. This is a better caching method for memory management because states are not deleted randomly.

Note that this method of an initial traversal of the state space can be applied to all current state space generation algorithms. For example [Pel94] presented an on–line model checker, by traversing the product of the state space and specification graphs. We also can initially traverse the product and calculate the revisiting degree of all nodes, saving space as shown in the following section.

4 Memory and Time Complexity

For analysis of memory and time complexity we distinguish between two stages: 1) The memory and time complexity of the algorithm that constructs the visible state graph (denoted VSG), and 2) The memory and time complexity of the algorithm for the model checking. In both cases the analysis is relative to the complexity of the algorithms that construct and model check the graph obtained by applying a partial order method (denoted POG). When we refer to memory complexity, our intention is internal memory. We assume that we have an unrestricted amount of external memory (used for the caching method). For the model checking itself, the savings is in checking a smaller model. Standard model checking of linear time temporal logic specifications has time and space complexity $O(|VSG| \cdot 2^{|\varphi|})$, whereas previously we had the same formula over the original state graph or the POG.

As for the preprocessing stage to compute the revisiting degree and then generate the visible state graph, the time complexity is the same as for existing methods for partial order reductions, namely $O(pe \cdot log(ps))$, where pe is the number of edges in the partial order graph that would be produced by that method alone, and ps is the number of nodes in that graph. As explained previously, our algorithm makes one additional traversal.

The partial order method we considered uses $O(m \cdot ps + log(m) \cdot pe)$ space in its original form, where m is the space needed for a single state.

For our algorithm, the memory complexity of the first DFS is $\mathrm{O}(m \cdot ss + ps)$ (where ss is the size of the stack). The data structures used are the search stack and the revisited hash table (using $O(ps)$ for the size of the hash table). The memory complexity for the subsequent reduction traversal is also $\mathrm{O}(m \cdot ss + ps)$. Here the data structures used are the search stack, the revisited hash table and the intermediate stages of the visible state graph construction. We use a caching method, therefore we bound by a constant the memory needed for the full states retained in intermediate stages. Our simulations show that the number of states needed at any one time is actually small, and the cache will not cause extraneous recomputation. The simulation was constructed using the high level language ICON. We implemented the algorithms that calculate the revisiting degree of a program's state graph and that generate the visible state graph. The ample set for the partial order method is calculated according to [HP94].

In one test, we simulated a leader election protocol in a unidirectional ring from [DKR82] and several alternative specification formulas. The algorithm uses a local variable max_i in each process to show its version of the maximal value.

We executed our algorithm on the state space of the protocol for 5 processes for 5 different specification formulas. In Figure 6, we compare for each formula the original size of the state graph (first states, and then edges), the state graph that was obtained only with the partial method, and the state graph that was obtained with our method (which includes the partial order method). The last column of the table presents the number of full nodes that were in memory at any time, in addition to the hash table.

The fifth formula was especially complex, namely: $\Diamond(((max_1 = 5) \wedge \neg((max_2 = 5) \vee \ldots (max_5 = 5))) \; U \; (((max_1 = 5) \wedge (max_2 = 5)) \wedge \neg((max_3 = 5) \vee (max_4 = 5) \vee (max_5 = 5))) \; U \; \ldots ((max_1 = 5) \wedge (max_2 = 5) \wedge (max_3 = 5) \wedge (max_4 = 5) \wedge (max_5 = 5))))$, i.e., the processes obtain the correct maximum in the fixed order 1,2,3,4,5. This formula's tableau state graph has on the order of 1000 states. In the model checking stage, multiplying this by the program's partial order state graph will result in a quarter of a million states, while multiplying this by the visible state graph will result in about 5,000 states. This formula is not satisfied by the protocol, because the order is actually random.

FORMULA	ORIG-S	ORIG-E	POG-S	POG-E	VSG-S	VSG-E	FULL
1	11099	68717	7030	21548	75	163	222
2	11099	68717	203	352	10	19	61
3	11099	68717	630	1373	42	98	81
4	11099	68717	723	1329	56	137	97
5	11099	68717	263	351	5	9	57

Fig. 5. Simulation results for leader election protocol.

In the table, for formula 1 our algorithm has reduced a state space of 11,099 states and 68,717 transitions to a state space of 75 states and 163 transitions, where the partial order method succeeded in reducing by less than one order of magnitude relative to the original. In addition to the hash table, only 222 full nodes were needed at any one time in the generation of the reduced graph. The reader can observe that the rest of the results are similarly impressive.

References

[DKR82] D. Dolev, M. Klawe and M.Rodeh. An O(nlogn) unidirectional distributed algorithm for extrema finding in a circle. *Journal of Algorithms*, volume 3, pages 245–260, 1992.

[GHP92] P. Godefroid, G.J Holzman and D. Pirottin. State space caching revisited. In *Proc. 4th International Conference on Computer Aided Verification*, LNCS 697, pages 178–191, Canada, June 1992.

[GW91] P.Godefroid and P.Wolper. A partial order approach to model checking. In *Proc. 6th Symposium on Logic in Computer Science*, pages 406–415, Amsterdam, July 91.

[Hol88] G.J.Holzmann. An improved protocol peachability analysis technique. *Software-Practice and Experience*, Vol 18(2), pages 137–161,February 1988.

[HP94] G.J. Holzmann and D. Peled. An improvement in formal verification. In *Proceedings FORTE 1994 Conference*, Switzerland, October 1994.

[KP92] S. Katz and D. Peled. Conditional independence using collapses. *Theoretical Computer Science*, volume 101, pages 337–359, 1992.

[LAM83] L. Lamport, What good is temporal logic? in: *Proc. IFIP 9th World Congress*, Paris, France (1983) 657–668.

[Pel94] D. Peled. Combining partial order reductions with on-the-fly model checking. In *Proc. 6th International Conference on Computer Aided Verification*, LNCS 818, pages 377–390, California, USA, June 1994.

[Val90] A. Valmari. A stubborn attack on state explosion. Formal methods in System Design 1, pages 297–322, 1992.

[VW86] M. Vardi and P.Wolper. An automata-theoretic approach to automatic program verification. In *Proc. 1st Symposium on Logic in Computer Science*, pages 322–331, Cambridge, June 1986.