High-Density Reachability Analysis [*]

Kavita Ravi Fabio Somenzi
Dept. of Electrical and Computer Engineering
University of Colorado at Boulder

Abstract

We address the problem of reachability analysis for large finite state systems. Symbolic techniques have revolutionized reachability analysis but still have limitations in traversing large systems. We present techniques to improve the symbolic breadth-first traversal and compute a lower bound on the reachable states. We identify the problem as one of density during traversal and our techniques seek to improve the same. Our results show a marked improvement on the existing breadth-first traversal methods.

1 Introduction

Coudert *et al.* [1] have shown that breadth-first traversal is more amenable to symbolic treatment than depth-first traversal, and hence can deal with sequential machines with many more states. Although quite successful, the symbolic methods developed so far (see, for instance, [2, 3, 4]) cannot complete the reachability analysis of many large finite state machines, because they require too much memory, or are computationally intensive.

Approximate traversal [5, 6] addresses this problem by computing a superset of the reachable states, thus enabling *conservative* verification. In this paper, we propose an alternative approach that can produce either certified exact reachability information, or a large sample of the entire reachable state set, thus enabling *partial* verification. Therefore, one can compute both a lower bound and an upper bound to the number of reachable states, and hence obtain a better characterization of the state space of a large sequential circuit.

Our approach targets at two important problems in the traversal of large sequential machines: the memory resources available for traversal and efficiency of traversal using symbolic techniques. Binary Decision Diagrams(BDDs) are extremely useful in representing characteristic functions of various sets in these symbolic algorithms. Since the efficiency of most BDD operations is dependent on the size of the BDDs, it is advantageous to keep their sizes(in terms of number of nodes) small. We claim that this advantage can be characterized as a *density* measure of BDDs, where *density* is defined as the ratio of minterms to nodes. We have, therefore, combined several techniques aimed at increasing density and present a reachability analysis algorithm that mixes breadth-first traversal and depth-first traversal.

The advantages of this mixed approach are two-fold; The mixed approach will, typically, reach more states with smaller memory resources as it is not constrained by a fixed sequence of states as in the breadth-first search. Even when the number of states are comparable, the mixed approach ventures farther away from the initial states, thereby producing a more uniform sampling of the reachable set. This helps find more errors in designs (subtle errors often require rather long sequences of events to be uncovered), and makes the results of partial verification more reliable in case of success.

Our combination of breadth-first and depth-first searches proceeds in breadth-first mode as long as the BDDs used in image computation are small. When they become too large, a small, dense subset is retained and used to proceed with traversal. Once we reach a stage where no new states are added we take the image of the entire reached set, which may be smaller towards the end, thus making this computation easier. This final computation is indeed critical from both the CPU time and the memory standpoint; hence, the algorithm may not be able to complete it, and return only a subset of the reachable states.

Previous attempts to mixed breadth-first/ depth-first traversal have focused on explicit search techniques [7], that are limited in the number of states they can explore, and on the combination of breadth-first traversal with *geometric chaining* [8]. Our approach is therefore the first fully general, symbolic solution to the problem. In the following section, we discuss the preliminaries. Section 3 contains the mixed breadth-first depth-first algorithm and techniques for subsetting. We report the results in Section 4.

2 Preliminaries

A *Finite State Machine* is defined as a 6-tuple $\langle I,O,S,\delta,\lambda,S^0\rangle$, where I is the input alphabet, O is the output alphabet, S is the (finite, non-empty) set of states, $\delta : S \times I \to S$ is the next state function, $\lambda : S \times I \to O$ is the output function, and $S^0 \subseteq S$ is the set of initial (reset) states.

Binary Decision Diagrams (BDDs) [9, 10] are directly acyclic graphs (DAGs) representing switching functions. Reduced Ordered BDDs are BDDs with the same ordering of variables appearing in different paths and are canonical representations of logic functions. Each internal node of a BDD is labeled with a variable v, and has two children, labeled T and E. The function f at any node, is evaluated as:

$$f = v \cdot f_T + v' \cdot f_E .$$

BDDs have proved to be very efficient data structures and are widely used in traversal applications.

The generalized cofactor, first proposed by Coudert *et al* [1], is an operator widely used in image computation. The generalized cofactor of a function f with respect to a function g is defined as:

$$f \downarrow g = f(\mu_g(x)),$$

where $\mu_g(x)$ is an apppropriate mapping of each minterm in the offset of $g(x)$ to a minterm in the on-set of $g(x)$.

Exact Traversal: Symbolic FSM traversal has been based so far on a breadth-first search (BFS) of the set of reachable states from a given set of initial states. At each iteration, the set of states, reachable from a given set (*From*) in one time-step, is computed. The set of new states, from this computation, are added to the reachable set (*Reached*) using the next state function. The algorithm terminates when the cumulative reached set attains a fixed point. All sets of states are represented as characteristic functions and BDDs are used to manipulate these characteristic functions.

Image Computation: We use the transition function method with the input splitting approach for image computation. The image is computed by first constraining δ

[*]This work was supported in part by NSF/DARPA grant MIP-9115432 and SRC contract 93-DJ-560.

with the set of states at each iteration ($\delta \downarrow$ *From*), and then computing the range of the constrained δ. There are two approaches to this image computation: based on input splitting and output splitting respectively. The efficiency of the transition function method lies in partitioning the components based on disjoint support and caching identical subproblems.

3 Improving Density

The main problem with symbolic BFS traversal is that it runs outs of memory when the size of BDDs representing the characteristic functions become too large. It has also been observed experimentally that in many cases the BDDs at completion are smaller than the intermediate ones. These observations suggest that the problem is due to the low density of the BDDs during traversal.

The sparse BDDs may represent different functions of the symbolic FSM traversal, one of which is *Reached*. The size of the *Reached* BDD grows as new states are added to it in each iteration. The BDD node size of *Reached* during traversal may show two different trends: Either it grows large and remains large or the intermediate size may be large but eventually the large number of minterms result in a smaller BDD. If the set of new states added at each iteration to *Reached* form a set of scattered minterms, the size of the BDD may explode. This problem may be especially alleviated by increasing the density of BDDs.

A second source of memory problems is image computation, which is carried out by computing the image of the constrained δ ($Image(\delta \downarrow From)$). We observed that, in many circuits, as BFS traversal progressed and the size of *From* increased, the constrained δ was several orders of magnitude larger than the original δ. The transition relation suffers from a similar inconvenience. A third set of sparse BDDs are the partial results of image computation.

One method of increasing the density of a given set of BDDs is to reorder the variables. Periodic reordering of the *Reached* BDD and dynamic reordering [11] during image computation, keeps the partial results small and increases density by lowering size. Our approach is to extract a dense subset of the set of states manipulated at a given iteration, such that the BDD for the subset is small. We target at exact traversal of large machines, failing which, we establish a lower bound. In this section, we discuss several techniques to improve density during traversal.

3.1 Combining Breadth-First and Depth-First Search

The modified BFS algorithm is shown in Figure 1. The machine is traversed with BFS traversal until the size of *From* exceeds a certain threshold; then, a dense subset is extracted and this subset of *From* is used to proceed with traversal. When no new states are produced, the sub-traversal may have reached the actual fixed point (the one that would be obtained during pure BFS traversal) or a *dead-end*, which arises from having discarded some states during the process of subsetting. Termination is checked by computing the image of *Reached*, which recovers those states that may have been discarded and not subsequently reached again.

A dense subset of *From* will have a positive impact on the size of the constrained δ as well as the size of *Reached*. Normally, adding a small BDD to that of *Reached* will not

```
SubsettingTraverse ($\delta, S^0$) {
        Reached = From = $S^0$;
        subsetting_traversal = 0;
        while (true) {
                To = Img ($\delta$,From);
                New = To $-$ Reached;
                if (New = 0) {
                        if (subsetting_traversal = 1)
                                To = Img ($\delta$, (Reached$\cup$New));
                        if (To = Reached) return Reached;
                        else New = To $-$ Reached;
                }
                From = New;
                if (size(From) > threshold) {
                        From = subset (From);
                        subsetting_traversal = 1;
                }
                Reached = Reached $\cup$ From;
        }
}
```

Figure 1: Breadth-First Traversal with Subsetting.

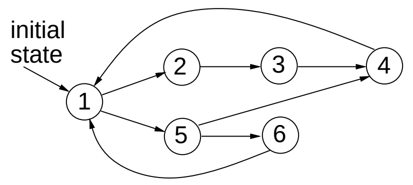

Figure 2: Example FSM.

increase the size of *Reached* as much as adding one large BDD. The advantage of subsetting lies in keeping the overall size and memory occupation low at all stages of the traversal. The threshold is chosen heuristically, trying to achieve fast image computation (by lowering the size of $\delta \downarrow From$) and small intermediate results, while not excessively increasing the number of iterations.

An example of traversal reaching a dead-end without reaching a fixed point is shown in Figure 2. At the second iteration of BFS traversal, *From* is $\{2, 5\}$. Suppose subsetting reduces it to $\{2\}$. Three more iterations lead to a dead-end where *Reached* $= \{1, 2, 3, 4\}$. Computing the image of *Reached* yields $\{1, 2, 3, 4, 5\}$. Traversal is therefore resumed with *From* $= \{5\}$, and proceeds until all six states are reached. Notice subsetting sometimes leads to overestimating the distance of a state from the reset states. In the example of Figure 2, State 6 is reached after five iterations, whereas, in pure BFS traversal, two iterations would suffice.

3.2 Computing Dense Subsets of BDDs

The problem of subsetting can be posed in the following way: Given a BDD for f with n nodes and a threshold $k < n$, find $g \leq f$ such that the BDD for g has $m \leq k$ nodes and that the number of minterms of g is maximum. If f has p minterms, there are 2^p choices for g. We therefore find it convenient to restrict ourselves to this modified version of the problem:
Subsetting Problem: Given a BDD F for f with n nodes and a threshold $k < n$, find a BDD G with $m \leq k$ nodes, such that:

1. G is obtained from F by eliminating some nodes, replacing all pointers to the eliminated nodes with pointers to the constant 0, and reducing the resulting graph.

2. The number of paths from the root of G to the constant 1 is maximum. The paths must be counted by assuming that each edge is indeed a multi-edge with multiplicity 2^d, where d is the difference between the indices of the

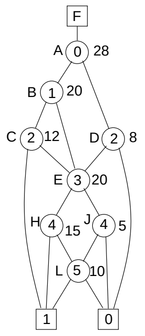

Figure 3: Shortcoming of Using the Number of Paths Through a Node for Subsetting.

two nodes connected by the edge. The index of the constant 1 is the largest variable index plus one.

The number of paths to the constant 1 in G is, according to the adopted definition, the number of minterms of the function g represented by G.

It is possible to compute in time linear in the size of F the number of paths from the root of F to the constant 1 that go through each node v of F. The number of paths through v gives the reduction in the number of minterms when v is eliminated (replacing all pointers to it by the constant 0). Hence, the numbers of paths through each node allow one to solve exactly the subsetting problem in linear time, for the case $k = n - 1$. However, for larger k, this would be suboptimal. Recomputing the number of paths through each node may result in better solutions but will still be expensive and suboptimal. Besides, this method may result in an unconnected graph.

Example: Consider the BDD of Figure 3, where each of the eight internal nodes is annotated with the number of paths from the root of F to the constant 1 that go through it. The best choice if we want a 7-node G is to remove node J. However, if we want a 3-node G and we discard the nodes with the fewest paths through them, J, D, L, C, and H, the resulting G, after reduction, is the constant 0. On the other hand, discarding D, E, H, J, and L gives, after reduction, a G with three nodes and eight minterms.

The two methods presented in the sequel try to address this problem in two different ways.

3.2.1 Heavy Branch Subsetting

The Heavy Branch method is based on the simple observation that a subset of a given BDD can be created by setting one cofactor of a node to the constant 0 [12] and retaining the other. This method uses the minterm and node count for each node in the BDD to guide the choice of the subset. The above two statistics for each node are computed and stored in a table as described in Figure 4. This method completes in 3 passes on the BDD.

```
Heavy_Branch (From, threshold) {
        minterm_count(v) = count_minterm (From);
        node_count(v) = differential_node_count (From,
                                        minterm_count(v));
        subset_size = bdd_size (From);
        subset = build_bdd (From, subset_size, threshold,
                        minterm_count(v), node_count(v));
        return (subset);
}
```

Figure 4: Heavy Branch Subsetting Algorithm.

1. In the first step, the *count_minterm* procedure counts the number of minterms for each node, v, in the BDD and stores it in *minterm_count*(v).

2. The *differential_node_count* procedure computes two measures for each node, v, in the BDD. One measure is the size of the DAG rooted at this node. The procedure first visits the branch with larger number of minterms (heavier child). By taking the heavier branch first, it is also possible to compute the *differential_node_count* of the lighter child of the node. The *differential_node_count* of the lighter child is defined as the number of nodes belonging exclusively to the lighter child i.e., a count that does not include nodes that are shared with the heavier child. Thus, each node, v, is labeled with its node count (size of BDD rooted at this node) and the *differential_node_count* of its lighter child, and this information is stored in *node_count*(v).

3. The next step of this procedure, starting at the root node, creates a subset of the given BDD by discarding (setting to the constant 0) the lighter child at each node. The size of the subset at each node v, created by eliminating the lighter child, is equal to the sum of the number of nodes on the path taken from the root node to v, and the node count of v less the *differential node* count of the discarded child. The process of discarding the lighter child continues until the size of the subset drops below the given threshold.

The advantage of this method is that it is fast and is linear in the size of the BDD being subset. In keeping the child with larger minterms, the subset of minterms retained is maximized. Pruning from the root node keeps a precise count of the size of the subset being created. The disadvantage of this method lies in the creation of a string of nodes at the top of the BDD, each of which has one child set to the constant 0. These nodes may cause a loss of recombination that may be acquired at no cost. Another drawback of this method is that it is dependent on the variable ordering in the BDD. The lighter branches from the top will always be eliminated, irrespective of the order of the variables present in the BDD. However, this dependence is offset to some extent by reordering.

Example: Consider the BDD in the Figure 5(a). The nodes of this graph, F, are annotated with the minterm count. Starting at the root node, A, the lighter children are eliminated until we reach the given threshold. If the threshold is 3, the resulting graph contains the nodes, A, B, and C, where the lighter children, D and E, are discarded. The resulting subset has 8 minterms vs. the 28 of F.

3.2.2 Short Paths Subsetting

The Short Paths method extracts the shortest paths between the root node and the constant 1(ONE). In a BDD, the shortest paths to ONE represent the largest sets of minterms, while the longest paths represent the smallest sets of minterms. Therefore, it is favorable to keep the shortest paths while creating a subset.

Consider a node v in the BDD. We define the *path_length* of this node as the sum of its shortest distances from the root and to ONE. The distance between two nodes in a BDD is

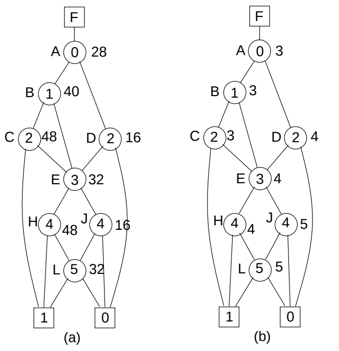

Figure 5: (a) Heavy Branch Subsetting (b) Short Paths Subsetting.

```
short_paths_subsetting (From, threshold) {
        root_dist(v) = find_shortest_root_dist (From);
        (ONE_dist(v), path_length(v)) =
                find_shortest_ONE_dist (From, root_dist(v));
        path_length_array(n) =
        find_shortest_path_lengths(root_dist(v),
                                            ONE_dist(v));
        max_path = find_max_path (path_length_array,
                                            threshold);
        subset = build_subset (From, path_length(v),
                                max_path, threshold);
        return (subset);
}
```

Figure 6: Short Paths Subsetting Algorithm.

defined as the number of edges between them. If v_{path_length} denotes the *path_length* of v, then any node, w, lying on the shortest path between the root node and ONE through v, has a w_{path_length} that is less than or equal to v_{path_length}. Thus, keeping the nodes with the shortest *path_length*s is equivalent to retaining the shortest paths in the BDD between the root node and ONE. The method involves choosing a maximum allowable *path_length* and building a subgraph with nodes that have *path_length*s less than or equal to the maximum allowable. Connectedness of the subgraph is ensured, since any node, v, with path length, v_{path_length}, has at least one child with *path_length* less than or equal to v_{path_length}, which provides a path to ONE. The Short Paths method completes in 3 passes of the BDD.

1. The first step is to find the shortest distance of each node from the root. This is requires in a breadth-first search of the BDD. Procedure *find_shortest_root_dist* stores the computed distance for each node in *root_dist*(v).

2. In the next step, *find_shortest_ONE_dist* computes the shortest distance of each node from ONE in a depth-first search of the BDD. Once the shortest distances from ONE are recorded, the *path_length* of each node can also be computed. The number of nodes labeled with each unique *path_length* are stored in *path_length_array*; if n is the number of variables, at most n different *path_length*s are possible.

3. The third step involves computing *max_path*, the maximum *path_length* such that the number of nodes with *path_length*s less than or equal to *max_path* sum up to the given threshold. The threshold may be such that not all nodes labeled with *max_path* are required.

4. The procedure, *build_subset*, generates a subset of *From* using the *path_length* information of each node. This procedure discards all nodes with *path_length* greater than *max_path* and retains as many complete paths as allowed be the threshold criterion. Choosing only a fraction of nodes of *max_path* may create an incomplete path i.e., retain nodes on a path that do not connect the root to ONE. In this case, the shortest path from this node to either ONE or any other node with *path_length* less than or equal to *max_path* is added to the subset. Some additional minterms are gained by adding this path with little overhead in terms of nodes (at most $(n-1)$ nodes beyond the threshold are to the added subset).

This method is designed to increase the density of subsets. Since paths of equal length contribute to the same number of minterms, and there is no relative advantage in choosing one over the other, this method performs best when the BDD has paths of varied lengths. However, since a subset of paths in the BDD are extracted, these paths may be disjoint. Consequently, a chosen set of paths may have very little sharing and this will have a negative impact on the density of the subset.

Example: Consider the example shown in Figure 5(b). The nodes of this graph F are annotated with *path_length*s. The nodes, A, B, C, are labeled with *path_length* 3, since the shortest path to ONE through them is A-B-C. Nodes, D, E, F, are labeled with *path_length* 4, where the shortest path they lie on is A-D-E-H. Nodes J and L are labeled 5 as they lie on the path A-B-E-J-L, which is 5 nodes long. If a 3-node graph is required, then nodes A, B, C are chosen to form the subset since they have the shortest *path_length*s.

Caching: Caching computed results, during image computation, can substantially improve the speed of traversal. Due to recombination in BDDs, the same subproblems could occur when taking different paths in the BDD. The input splitting approach benefits from this recombination especially as the subproblems get small.

Dynamic Reordering during Image Computation: Reordering of variables increases the density of BDDs in traversal by reducing their size and the speedup due to the smaller BDDs often offsets the time spent reordering. It has also been observed experimentally that the best order for δ is often not the best one for traversal. Periodic reordering of variables during traversal is required to keep the size small at all times. Sometimes, traversal runs out of memory during image computation since the partial results are large. The incorporation of dynamic reordering [11], during image computation, has shown dramatic improvements in being able to overcome the size increase that occurs when traversing some circuits.

4 Experimental Results

In this section, we present experimental results on the improved techniques for traversal. we have integrated our techniques into the symbolic FSM traversal package, *verif* [4]. All experiments were conducted on a 275MHz DEC Alpha workstation. We set a time limit of 22000 CPU seconds on the experiments and a memory limit of 440M.

Table 1 contains results for some of the ISCAS89 benchmark circuits and a data link controller circuit, `controller`. The third row of this table shows results for the `s5378` benchmark optimized with VERITAS [13]. The optimization was done by removing redundant latches using approximate

Example	Statistics	BFS	Dyn. Reord.	Cache & Dyn.	Subsetting	
					Heavy Branch	Short Paths
controller (172 latches)	Time	Mem. Out	> 22000s	2593s		
	Reached states	2.85e+45	2.85e+45	2.85e+45		
	Reached nodes	2503632	97015	40175		
	iterations	1	1	6998		
s5378 (164 latches)	Time	Mem. Out	>22000s	>22000s	>22000s	>22000s
	Reached states	1.27e+09	1.73e+12	1.73e+12	**3.40e+12**	**1.13e+15**
	Reached nodes	3177637	14240	8014	56853	38066
	iterations	2	3	3	115	13
s5378 opt (121 latches)	Time	Mem. Out	>22000s	>22000s	>22000s	>22000s
	Reached states	4.57e+07	1.67e+14	1.35e+16	**1.66e+17**	**1.08e+17**
	Reached nodes	873004	10672	39053	13319	23686
	iterations	2	4	12	147	56
s9234 (228 latches)	Time	>22000s	>22000s	>22000s		
	Reached states	7.82e+08	2.06e+09	1.32e+12		
	Reached nodes	86257	2006	1944		
	iterations	392	454	1284		
s1423 (74 latches)	Time	Mem. Out	Mem. Out	Mem. Out	>22000s	>22000s
	Reached states	4.8e+08	7.99e+09	4.8e+08	**5.56e+11**	**1.34e+11**
	Reached nodes	792380	590812	134765	209806	109137
	iterations	9	11	10	232	353

Table 1: Experimental Results of Subsetting Traversal.

traversal which reduced the number of latches to 121. These circuits are reasonably large and have proved intractable using BFS traversal. Column 2 shows statistics of the breadth-first traversal. Most circuits ran out of memory during image computation. Column 3 reports results of BFS traversal with dynamic reordering. Column 4 reports results on the incorporation of the cache during image computation. Columns 5 and 6 report traversal results with the subsetting techniques.

As indicated by the table, the subsetting techniques are very effective and produce a larger *Reached* set than any other column. In the case of the `controller` example and `s9234`, the *From* BDDs are very small and subsetting was not required. As reported in [13], the upper bound for `s5378 opt` and `s5378` is 2.95e+17. With subsetting, in the case of `s5378 opt`, we have a *Reached* set that has more than half the number of states in the upper bound. We ran the `s1423` example for longer than the time limit specified on the above results and obtained a reached set of size 1.67e+14 while the upper bound reported in [13] is 6.25e+18. In general, the subsetting method of *Short Paths* produced very dense subsets and the ratio of the densities of the subset BDD to the original BDD was sometimes as high as 5.35. The table reflects the impact of these high densities on the reduction of the number of iterations to reach the same number of states.

5 Conclusions and Future Work

In this paper, we have presented techniques that increase the density of traversal. These results can be viewed as a practical improvement over simulation in terms of number of states sampled per CPU second. The subsetting techniques have also proved very effective in furthering traversal and establishing a lower bound of the reachable set. Compared to the symbolic BFS traversal, our method is well behaved, in terms of memory occupation and does better in terms of the number of reachable states. In some cases, we can combine our results with approximate traversal and prove tight bounds on the number of reachable states of the machine. The subsetting techniques can be extended to any fixed point computation problem that behaves like FSM traversal. In the future, we propose to study the applications of these techniques to approximate traversal and model checking.

Acknowledgemnts We would like to thank Carl Pixley and Bernard Plessier for providing us with examples.

References

[1] O. Coudert, C. Berthet, and J. C. Madre, "Verification of sequential machines using boolean functional vectors," in *Proc. IFIP International Workshop on Applied Formal Methods for Correct VLSI Design* (L. Claesen, ed.), (Leuven, Belgium), pp. 111–128, Nov. 1989.

[2] J. R. Burch, E. M. Clarke, D. E. Long, K. L. McMillan, and D. L. Dill, "Symbolic model checking for sequential circuit verification," *IEEE Transactions on Computer-Aided Design*, vol. 13, pp. 401–424, Apr. 1994.

[3] H. Cho, G. D. Hachtel, S.-W. Jeong, B. Plessier, E. Schwarz, and F. Somenzi, "ATPG aspects of FSM verification," in *Proc. ICCAD* , pp. 134–137, Nov. 1990.

[4] S.-W. Jeong, B. Plessier, G. D. Hachtel, and F. Somenzi, "Variable ordering and selection for FSM traversal," in *Proc. ICCAD* , (Santa Clara, CA), pp. 476–479, Nov. 1991.

[5] H. Cho, G. D. Hachtel, E. Macii, B. Plessier, and F. Somenzi, "Algorithms for approximate FSM traversal based on state space decomposition," in *Proc. DAC*, (Dallas, TX), pp. 25–30, June 1993.

[6] H. Cho, G. D. Hachtel, E. Macii, M. Poncino, and F. Somenzi, "A structural approach to state space decomposition for approximate reachability analysis," in *Proc. ICCD* , (Cambridge, MA), pp. 236–239, Oct. 1994.

[7] D. L. Dill, A. J. Drexler, A. J. Hu, and C. Han Yang, "Protocol verification as a hardware design aid," in *Proc. ICCD* , (Cambridge, MA), pp. 522–525, Oct. 1992.

[8] A. Ghosh and S. Devadas, "A mixed depth-first/breadth-first technique for sequential logic verification," in *IWLS* , (MCNC, Research Triangle Park, NC), May 1991.

[9] R. E. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Transactions on Computers*, vol. C-35, pp. 677–691, Aug. 1986.

[10] K. S. Brace, R. L. Rudell, and R. E. Bryant, "Efficient implementation of a BDD package," in *Proc. DAC* , (Orlando, FL), pp. 40–45, June 1990.

[11] R. Rudell, "Dynamic variable ordering for ordered binary decision diagrams," in *Proc. ICCAD* , (Santa Clara, CA), pp. 42–47, Nov. 1993.

[12] J. Jain, J. Bitner, D. S. Fussell, and J. A. Abraham, "Functional partitioning for verification and related problems," in *Brown/MIT VLSI Conference*, Mar. 1992.

[13] H. Cho, G. D. Hachtel, E. Macii, M. Poncino, K. Ravi, and F. Somenzi, "Approximate finite state machine traversal: Extensions and new results." Presented at IWLS95, Lake Tahoe, CA., May 1995.