

Inductively Finding a Reachable State Space Over-Approximation

Michael L. Case Alan Mishchenko Robert K. Brayton
Department of EECS, University of California, Berkeley
{casem, alanmi, brayton}@eecs.berkeley.edu

Abstract— We present an algorithm to find an over-approximation of the set of reachable states in a finite state machine, and we present highly tuned data structures to make this method efficient. The algorithm works by inductively proving implications of the form $a \Rightarrow b$ between pairs of nodes in the logic network. Because each implication proved is guaranteed to hold in every reachable state, the conjunction of the implications forms an over-approximation of the reachable state set. This over-approximation becomes tighter as more implications are proved, providing the user with a runtime vs. quality trade-off. Experimental results show that this method is robust and scalable.

I. INTRODUCTION

Given a finite state machine, many applications require exploration of the subset of states reachable from an initial set of states. Some of these application areas are:

Combinational optimization may be performed using unreachable states as external don't cares. This effectively modifies the circuit in states that are known to be unreachable.

Sequential redundancies can be found and eliminated. It is known that in general there are wires that appear redundant under all reachable states but may be irredundant in an unreachable state.

Formal Verification, either sequential equivalence checking or property checking, can be made more efficient by reducing its search space to an over-approximation of the set of reachable states.

Conventional reachability analysis is practical only for small designs. It explores the state space in a breadth-first manner, starting from the initial state. Traditionally, binary decision diagrams (BDDs) are used to represent the set of states that have been visited. While this approach is very fast when run on small designs, it is subject to the BDD explosion problem, resulting in unpredictability and failure to terminate on designs of more than 100 or so latches. Since this approach is not rugged, it is not used in industry. Thus, the potential benefits from knowing the reachable state set motivate the exploration of methods that do state reachability faster and more reliably for large designs.

We propose a fast method to over-approximate the set of reachable states. Given the logic functions a and b at the output of two nodes in the network, we can inductively prove that $a \Rightarrow b$ in every reachable state. We prove a large candidate set of implications simultaneously, rejecting implications that fail our inductive hypothesis as we iterate. When a fixed point is reached, the remaining implications hold in every reachable state. Since the reachable state set is contained in the space of states for which each implication holds, the conjunction of the implications presents an over-approximation to the reachable state set. This concept is illustrated in Figure 1.

Our approach has the property that every additional implication proved tightens the over-approximation. Therefore the computation

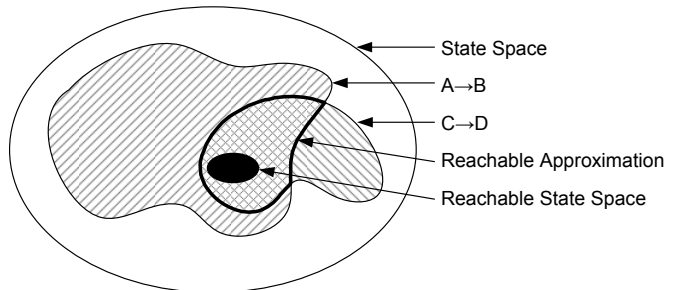


Fig. 1. The implications $A \Rightarrow B$ and $C \Rightarrow D$ each contain the reachable state space, and their product gives us an over-approximation of the reachable state set.

can be terminated after proving only a subset of the possible implications and still have a usable, conservative over-approximation. This property makes our method more robust than BDD-based approaches which must always run to completion before any information is obtained.

II. RELATED WORK

Several methods have been given to approximate reachability analysis. Most methods simplify BDD-based reachability analysis. Ravi [3] proposed a method to simplify the BDD representing the set of reachable states during the progression of the reachability algorithm. Several others [2], [5], [6] proposed methods to partition the state transition relation before it is converted to a BDD. These methods result in a BDD-based implementation that scales better, yet still suffers from the fundamental BDD explosion problem and hence failure to produce a usable result in many designs. Our method is based on Boolean satisfiability (SAT), which allows it to scale better than an optimized BDD-based method.

McMillan [7] presents a way to use length- k clauses to approximate don't care sets and to accelerate don't care computation in combinational circuits. In contrast, we find implications between nodes (essentially 2-literal clauses), which hold in every reachable state. This allows the use of a simpler inductive proof technique, rather than McMillan's trie quantification approximation.

Van Eijk [8] proposed an inductive technique to discover equivalent nodes between two similar designs. This was proposed to speed up sequential equivalence checking. Bjesse [9] improved on van Eijk's results by strengthening the inductive hypothesis. We focus on synthesis, a domain where finding many equivalent nodes would be rare but implications may be abundant. We tailor our data structures and SAT methods for implications and provide a framework to trade runtime for quality of results.

III. FINDING IMPLICATIONS TO APPROXIMATE THE REACHABLE STATE SET

Our method is outlined in the following pseudocode:

```

chose an initial candidate implication window;
while (!timeout) {
  prove inductively that
    a subset of the candidate implications hold
    in every reachable state;
  widen the candidate implication window;
}
construct the reachable state space approximation;

```

The details of how candidate implications are proved, how implications are tracked in the current working set, how candidate windowing and timeouts are used, and how the reachable state set over-approximation is built follow below.

A. Proving a Set of Implications

Suppose a set of candidate implications $I = \{(a, b) \mid a \Rightarrow b\}$ is given. A technique called k -step induction is used to prove that a subset of I holds in every reachable state.

In the base case, all states reachable in k or less transitions from the initial state are determined, and only those elements of I are kept which hold in all these states.

The inductive hypothesis is that if an implication holds for a sequence of k consecutive states, then it should also hold in every next state reachable from the last state in the sequence. During the induction step, if any member of I fails to hold in this context, it is discarded, and induction is run on the reduced I set.

On convergence, I has been refined to a subset of implications which have been proved (by induction) to hold in every reachable state. The important point is that this has been proved by induction rather than reachability analysis.

Example: Suppose $k = 2$ and the candidate implication set is $I = \{a \Rightarrow b, b \Rightarrow c, c \Rightarrow d, d \Rightarrow e\}$ where $a, b, c, d,$ and e are signals at the outputs of a selected set nodes in the network. We unroll the circuit $k = 2$ times by adjoining 2 consecutive copies of the circuit. The first frame represents an arbitrary circuit configuration, and the second represents every state reachable in one step. We use the notation $(\cdot)_n$ to indicate that the argument implication holds in the n 'th frame. The base case involves checking that the following expression holds in every initial state and for all possible assignments to primary inputs.

$$(a \Rightarrow b)_1 \wedge (a \Rightarrow b)_2 \wedge (b \Rightarrow c)_1 \wedge (b \Rightarrow c)_2 \wedge (c \Rightarrow d)_1 \wedge (c \Rightarrow d)_2 \wedge (d \Rightarrow e)_1 \wedge (d \Rightarrow e)_2$$

Suppose $(d \Rightarrow e)_2$. Then there is a state reachable in one step from an initial state for which this implication does not hold, and $d \Rightarrow e$ is removed from the candidate set of implications. In the inductive step, no assumption is made about the state of the first frame, i.e. all latches and primary inputs are free variables. We then check

$$\left[(a \Rightarrow b)_1 \wedge (b \Rightarrow c)_1 \wedge (c \Rightarrow d)_1 \right] \Rightarrow \left[(a \Rightarrow b)_2 \wedge (b \Rightarrow c)_2 \wedge (c \Rightarrow d)_2 \right]$$

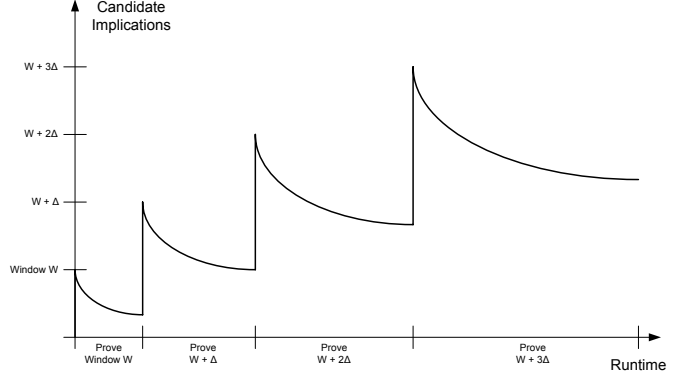


Fig. 2. Our hierarchical proof technique.

Suppose a set of inputs is found such that the left hand side holds, but $(c \Rightarrow d)_2$. Since the left hand side characterizes the current approximation to the set of reachable states, the above says that in a state that we currently assume to be reachable, there is a next state where $c \not\Rightarrow d$. Therefore $c \Rightarrow d$ does not hold in every reachable state, and we remove it from the candidate set of implications. If no other implications are disproved then $\{a \Rightarrow b, b \Rightarrow c\}$ is an invariant of every reachable state and hence characterizes an over-approximation of the set of reachable states.

B. Window Proof Technique

We have developed a windowing technique to limit the number of candidate implications being proved at any one time and to provide a hierarchical technique that gives an ever tightening reachable state set approximation. It is this windowing technique that enables the runtime vs. quality of results trade-off.

Any candidate set of implications is pruned until every implication in the set holds in all approximate reachable states. Until this fixed point is reached, we cannot say that the conjunction of the implications gives an over-approximation. It is desirable be able to terminate early with useful partial results. Thus we use an ever widening window. At each step, only candidate implications in the window are assumed and pruned until a fixed point is reached for that set. Then the window is widened and a proof of a new set of implications begins. At any point, the algorithm can be interrupted, and we know that all implications proved up to this point are valid.

The implications in the fixed point of each window hold in every reachable state and so will continue to hold once the candidate implication window is widened. Therefore these implications are assumed to hold in the next windowing step. This helps to speed up the next proof by excluding known-unreachable states from the left-hand side of the inductive hypothesis. However, implications which fail to hold in a particular window must be considered as candidates in the next window iteration because we may learn more about the reachable state space in the next window. This may cause a previously false implication to appear true now.

Example: In Figure 2, we start with an initial window W and a set of candidate implications. This is pruned until a fixed point is reached, and then the window is widened to $W + \Delta$. This increases the candidate implication set, and the inductive proof technique is run again. On interruption at any time during this

proof of the $W + \Delta$ implications, only the fixed point reached using the W candidates is valid.

An open problem is how to best choose the candidate implication window. This depends heavily on the target application. For reachability used with combinational optimization, a random candidate window seems to work best. For property verification, checking a safety property is the same as showing that a set of bad states is not reachable. In such a context, it makes sense to choose the window such that only implications that fail to hold in at least one bad state are considered. If successful in proving an implication, then the state space over-approximation, i.e. the conjunction of the implications (Section III-D), will evaluate to 0 on a set of bad states, which implies that these bad states are unreachable.

C. The Implication Graph

Throughout the algorithm, we maintain a set of candidate implications. We have developed a set of algorithms and highly tuned data structures to minimize this set as much as possible.

The implications can be viewed as forming a directed graph. Its nodes are the nodes of the original logic network, and each directed edge represents one implication. There is a graph theoretic procedure that can minimize this implication graph by representing only a subset of implications, from which all others can be derived. This minimization allows for a dramatic compression of the candidate set. In practice, we observe that by paying the overhead to compress the graph, the candidate set can be reduced to less than 5% of its original size (Section IV-E), leading to a significant speed-up of the overall algorithm.

The key property is transitivity; if $a \Rightarrow b$ and $b \Rightarrow c$ then $a \Rightarrow c$. Hence $a \Rightarrow c$ need not be stored. On an implication graph, this reduction is equivalent to removing any implication $x \Rightarrow y$ if y is reachable from x on an alternate path. Aho et. al. [10] use this concept in the following definition:

Definition 1 (Transitive Reduction): Given a directed graph G , the transitive reduction $r(G)$ is a directed graph satisfying the following properties:

- 1) The vertex set of G is equal to the vertex set of $r(G)$.
- 2) There is a directed path from vertex u to vertex v in G if and only if there is a directed path from u to v in $r(G)$.
- 3) There is no graph with fewer edges than $r(G)$ satisfying the above conditions.

Aho shows that the transitive reduction of a directed acyclic graph is unique. In the case of a cyclic graph, one must find all strongly connected components, join the nodes in the component by a single cycle, and then use one member of the cycle as a representative. The graph of all representatives is called the *condensed graph*. It is acyclic and hence can be reduced uniquely. Therefore the transitive reduction of a general directed graph is unique in cardinality.

Example: Figure 3 illustrates transitive reduction. To reduce the top graph, the strongly connected components are identified as $\{A, B, C\}$ and $\{D, E, F\}$. Each component is replaced by a simple cycle joining all nodes in the component. B and E are designated as the component representatives in the condensed graph. The condensed graph is acyclic, and its transitive reduction can be found by greedily removing edges $u \Rightarrow v$ for which there exists an alternate “*reducing path*” from u to v . This produces the transitively reduced bottom graph in the figure. Note that in general the edge set of the reduced

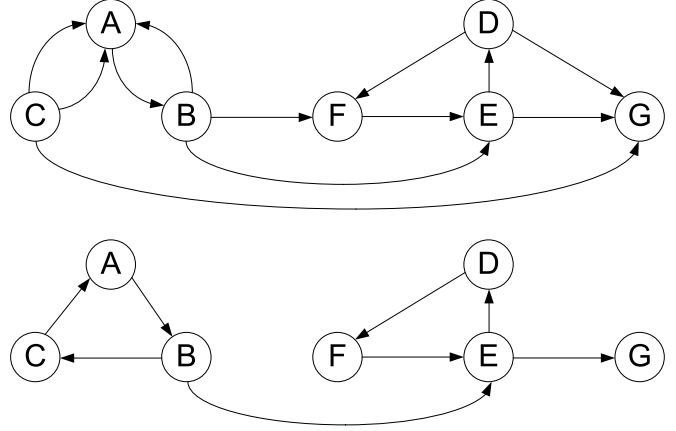


Fig. 3. An example directed graph and its transitive reduction.

graph is not a subset of the edge set of the original graph. The reader can verify that the graph shown does indeed satisfy the definition of a transitive reduction.

In this application, we require two graph algorithms. Given a transitively reduced graph, the transitive reduction must be maintained under the following operations:

Vertex addition. When the candidate implication window is widened, vertices and their associated edges must be added to the graph. Adding redundant edges must be avoided, and we must efficiently identify when a reducing path is introduced for a pre-existing edge.

Edge deletion. When the candidate implication set is refined, edges from the implication graph must be removed. In general, there may be a number of edges not represented in the transitive reduction because they were redundant, but removing a single edge could make them irredundant. These must be identified and placed back into the graph. For example, if $a \Rightarrow b \Rightarrow c$ then $a \Rightarrow c$ is a redundant edge that will not be present in the graph. However, upon removing $b \Rightarrow c$, $a \Rightarrow c$ becomes irredundant if there is no other path from a to c and so this edge is added to the transitive reduction. Thus, removing an edge from a transitive reduction may increase the size of the graph.

The graph algorithms developed for this work are polynomial in time complexity and dramatically reduce the size of the implication graph. There is a runtime penalty that must be paid in order to always maintain the transitive reduction, but in practice this is well worth the effort. More information about these algorithms can be found in [1].

D. Building the Reachable State Space Approximation

If P is the set of primary inputs to the logic network and I is the set of proved implications then the over-approximation to the set of reachable states is given by:

$$\bigvee_{x \in P} x. \bigwedge_{(a \Rightarrow b) \in I} (a \Rightarrow b)$$

Because every implication holds in all reachable states, we may safely universally quantify out all primary inputs to get an expression over the latches only. The conjunction of expressions that

TABLE I
 RUNTIME AND QUALITY OF RESULTS COMPARISON BETWEEN BDD-BASED REACHABILITY AND OUR METHOD.

Benchmark	Circuit Statistics ¹			Runtime (sec)		Reachable States (%)	
	Levels	Latches	Nodes	Exact Reach.	Approx. Reach.	Exact Reach.	Approx. Reach.
s27	5	3	8	0.07	0.38	75.00	75.00
s208	9	8	71	0.10	1.05	100.00	100.00
s298	9	14	100	0.13	0.96	1.33	3.10
s344	13	15	104	0.18	1.15	8.01	70.73
s349	13	15	104	0.20	1.10	8.01	70.73
s382	12	21	132	0.14	10.74	0.42	1.68
s400	13	21	139	0.18	10.12	0.42	1.44
s444	14	21	143	0.22	13.14	0.42	1.95
s641	25	19	146	0.39	5.34	0.29	0.42
s713	25	19	149	0.44	5.43	0.29	0.42
s420	11	16	157	1.76	35.08	100.00	100.00
s386	10	6	166	0.08	3.29	20.31	20.31
s526n	9	21	185	0.24	19.96	0.42	1.50
s526	9	21	186	0.17	22.62	0.42	1.88
s510	11	6	213	0.11	5.92	73.44	73.44
s820	14	5	331	0.12	69.12	78.12	78.13
s838	15	32	333	3599.51	103.47	-	100.00
s832	14	5	343	0.10	69.12	78.12	78.13
s1423	54	74	459	3605.45	82.08	-	7.13
s1196	19	18	477	0.37	26.16	1.00	56.84
s1238	21	18	525	0.39	32.19	1.00	56.84
s1488	15	6	662	0.13	91.35	75.00	75.00
s1494	15	6	671	0.13	100.34	75.00	75.00
s5378	17	164	1299	3599.65	130.04	-	75.52
s9234	32	211	1791	3599.75	131.74	-	40.17
s13207	34	669	2621	3599.82	136.08	-	0.12
s15850	45	597	3357	3599.63	129.91	-	100.00
s38417	31	1636	9063	3599.70	146.28	-	65.79
s38584	36	1452	11644	3599.76	147.94	-	22.32

¹ Numbers reported after some combinational preprocessing.

hold in reachable states provides a tighter approximation to the reachable states than any of its constituents. We move the universal quantification outside of the product for convenience.

In practice, this quantification is expensive, but is only required if the over-approximation must be expressed in terms of the latches. Another way is to simply assert the proved implications as additional SAT clauses in an application. Then there is no need to project the implications onto the latch space.

IV. EXPERIMENTAL RESULTS

Our reachability over-approximation algorithm was implemented in the ABC [17] logic optimization and verification framework, and utilizes the SAT solver MiniSat 1.14 [19]. In this section, we summarize 7 separate experiments using this framework.

A. Comparison With BDD-Based Reachability

Table I compares the performance of our algorithm against a BDD-based reachability algorithm implemented in MVSIS [18]. The ISCAS89 benchmarks were run through both systems on a 3GHz Pentium 4 machine. The BDD method was given a time limit of 1 hour, and our method was given a time limit of 60 seconds. The last two columns in the table show the percentage of states that are estimated to be reachable relative to the total state space. When our method reports that more states are reachable than the exact method reports, the difference is the reachable state space over-approximation. Therefore throughout these experiments, lower reachability percentages are better. It is interesting that our approximation manages to find a reachable state space approximation even

for large designs where BDD-based techniques fail. Furthermore, it finds approximations even given short time limits.

B. Ideal Bounds on Our Method

Table II illustrates the limitation of only using implications to approximate the set of reachable states, and it also shows a way to strengthen our results. In this work, the reachable state set is approximated by those states which can satisfy some conjunction of implications. It is not obvious that such a conjunction could ever completely characterize the reachable state set. Furthermore, implications are limited to those between nodes pre-existing in the design. A design may lack nodes conducive to implications, which would limit our method. To test the effectiveness of using implications, we found the exact set of reachable states for the small benchmarks and then projected this set onto the set expressible as a conjunction of implications. If R is the characteristic function for the exact set of reachable states, it can be projected onto the set expressible with implications by:

$$\bigwedge \{a \Rightarrow b \mid (R \Rightarrow (a \Rightarrow b)) = 1\}$$

The states that satisfy this expression give an over-approximation of the reachable state set that is the best allowed by a conjunction of implications. Table II shows this ideal lower bound. The difference between this projected reachability and exact reachability is the over-approximating bias that any implication-based method must have. The table illustrates how close our method comes to this fundamental lower bound on the over-approximation.

TABLE II
COMPARING k -STEP INDUCTION RESULTS TO THE IDEAL IMPLICATION RESULTS.

Benchmark	Circuit Statistics			Exact Reachability Projection ¹		Approximate Reachability (%), Sweeping k ^{1,2}		
	Levels	Latches	Nodes	Exact Reach.	Projected to Imps.	$k = 2$	$k = 3$	$k = 4$
s27	5	3	8	75.00	75.00	75.00	75.00	75.00
s208	9	8	71	100.00	100.00	100.00	100.00	100.00
s298	9	14	100	1.33	3.05	3.10	3.10	3.10
s344	13	15	104	8.01	70.53	70.73	70.73	70.73
s349	13	15	104	8.01	70.53	70.73	70.73	70.73
s382	12	21	132	0.42	1.47	1.68	1.68	1.68
s400	13	21	139	0.42	1.47	1.44	1.44	1.44
s444	14	21	143	0.42	1.51	1.95	1.95	1.95
s641	25	19	146	0.29	0.42	0.42	0.42	0.42
s713	25	19	149	0.29	0.47	0.42	0.42	0.42
s420	11	16	157	100.00	100.00	100.00	100.00	100.00
s386	10	6	166	20.31	20.31	20.31	20.31	20.31
s526n	9	21	185	0.42	1.44	1.50	1.50	1.50
s526	9	21	186	0.42	1.44	1.88	1.88	1.88
s510	11	6	213	73.44	73.44	73.44	73.44	73.44
s820	14	5	331	78.12	78.12	78.12	78.12	78.12
s832	14	5	343	78.12	78.12	78.12	78.12	78.12
s1423	54	74	459	-	-	7.13	7.13	7.13
s1196	19	18	477	1.00	15.62	56.84	56.84	56.84
s1238	21	18	525	1.00	15.62	56.84	56.84	56.84
s1488	15	6	662	75.00	75.00	75.00	75.00	75.00
s1494	15	6	671	75.00	75.00	75.00	75.00	75.00
s9234	32	211	1791	-	-	75.52	12.55	12.79
s13207	34	669	2621	-	-	0.12	₃	₃
s15850	45	597	3357	-	-	100.00	100.00	100.00
s38417	31	1636	9063	-	-	65.79	33.64	₃
s38584	36	1452	11644	-	-	22.32	78.58	78.53

¹ All numbers reported as the percentage of states reachable relative to 2^n where n is the number of latches in the design.

² The runtime of our method was limited to 2 minutes.

³ We use a BDD-based method to count the number of reachable states. Our method succeeded here, but we were unable to build the BDD for what was computed.

C. Examining k -Step Induction

Table II also shows the effect of increasing k in “ k -step induction” (Section III-A). This reduces the occurrence of implications being spuriously disproved and should provide better results, as explained in [9]. As Table II shows, in most cases the over-approximation gets tighter as k increases. On the other hand, the size of the SAT problem increases with k , slowing down the overall algorithm. Under a time limit, this can cause less implications to be proved. Since the benefit obtained by increasing k above 1 appears negligible, for all other experiments, we fix $k = 2$.

D. Examining the Runtime Vs. Quality Trade-Off

The runtime versus quality of results trade-off is a key contribution of this paper. In Table III, the maximum processing time allowed in our implementation is varied. As the processing time is increased, the candidate window of implications can grow larger. Thus more implications will be found, and a tighter approximation of the reachable state set will be obtained. The table illustrates a runtime versus quality trade-off of our method, which gives the user more control. Note that in all other experiments, the time limit was limited to 60 seconds.

E. Examining the Transitive Reduction

One of the contribution of this paper is the use of the transitive reduction to compact the implication graph. In Table IV we examine the effectiveness of this transitive reduction. There are two ways to store the set of implications in the candidate window: naively storing

each implication in a linked list, or building a transitively reduced implication graph. The table uses several metrics to compare these two implementations:

Runtime: We observe that the runtime is vastly improved by using the transitive reduction. This represents a balance between working with a reduced set of implications and the overhead to maintain this reduction.

Implications Per SAT Call: The number of implications involved in each SAT call has been reduced greatly. This simplifies the formulation of Section III-A and lessens our dependence on the SAT solver.

Peak Memory Usage: The transitive reduction implementation only needs to store the reduced implication graph. There is overhead in the graph data structures, but in general this is much less than the memory required to store every implication explicitly.

Overall the transitive reduction is valuable in allowing our technique to scale to large designs.

F. Examining the Application to Synthesis

The introduction mentioned that the reachable state information can be used by a combinational optimizer by expressing unreachable states as external don’t cares (EXDCs). In Table V, we used two sets of unreachable states in this optimization: one is the exact set obtained from the BDD-based implementation, and the other is that obtained by our algorithm in 2 minutes with $k = 2$. Table V shows three meaningful figures:

TABLE III
COMPARISON OF THE QUALITY OF RESULTS OVER TIME.

Benchmark	Circuit Statistics			% Reachable After a Fixed Amount of Processing Time				
	Levels	Latches	Nodes	20 sec	40 sec	60 sec	80 sec	100 sec
s27	5	3	8	75.00	75.00	75.00	75.00	75.00
s208	9	8	71	100.00	100.00	100.00	100.00	100.00
s298	9	14	100	3.10	3.10	3.10	3.10	3.10
s344	13	15	104	70.73	70.73	70.73	70.73	70.73
s349	13	15	104	70.73	70.73	70.73	70.73	70.73
s382	12	21	132	1.68	1.68	1.68	1.68	1.68
s400	13	21	139	1.44	1.44	1.44	1.44	1.44
s444	14	21	143	1.95	1.95	1.95	1.95	1.95
s641	25	19	146	0.42	0.42	0.42	0.42	0.42
s713	25	19	149	0.42	0.42	0.42	0.42	0.42
s420	11	16	157	100.00	100.00	100.00	100.00	100.00
s386	10	6	166	20.31	20.31	20.31	20.31	20.31
s526n	9	21	185	1.50	1.50	1.50	1.50	1.50
s526	9	21	186	1.88	1.88	1.88	1.88	1.88
s510	11	6	213	73.44	73.44	73.44	73.44	73.44
s820	14	5	331	78.12	78.12	78.12	78.12	78.12
s838	15	32	333	100.00	100.00	100.00	100.00	100.00
s832	14	5	343	78.12	78.12	78.12	78.12	78.12
s1423	54	74	459	7.15	7.15	7.13	7.13	7.13
s1196	19	18	477	56.84	56.84	56.84	56.84	56.84
s1238	21	18	525	56.84	56.84	56.84	56.84	56.84
s1488	15	6	662	75.00	75.00	75.00	75.00	75.00
s1494	15	6	671	75.00	75.00	75.00	75.00	75.00
s9234	32	211	1791	40.95	40.17	40.17	34.79	34.79
s13207	34	669	2621	0.17	0.17	0.12	0.00	0.00
s15850	45	597	3357	100.00	100.00	100.00	100.00	100.00
s38417	31	1636	9063	82.98	81.25	65.79	62.29	62.29
s38584	36	1452	11644	84.67	84.67	22.32	- ¹	- ¹

¹ We use a BDD-based method to count the number of reachable states. Our method succeeded here, but we were unable to build the BDD for what was computed.

TABLE IV
ILLUSTRATING THE BENEFITS OF THE TRANSITIVE REDUCTION.

Design	Runtime (sec)			Average Implications Per SAT Call			Peak Memory Usage (MB)		
	Reduced	Naive	Ratio (%)	Reduced	Naive	Ratio (%)	Reduced	Naive	Ratio (%)
s27	0.99	0.59	167.80	31.5	44.5	70.79	15.16	17.18	88.24
s208	1.52	2.93	51.88	284.02	2238.15	12.69	19.4	18.4	105.43
s298	1.39	3.02	46.03	361.16	1858.38	19.43	16.74	22.6	74.07
s344	2.17	2.02	107.43	366.96	1350.76	27.17	20.23	20.79	97.31
s349	1.59	2.33	68.24	366.96	1350.76	27.17	17.05	18.75	90.93
s382	14.15	103.25	13.70	451.21	12684.41	3.56	22.61	30.33	74.55
s400	14.17	100.03	14.17	461.12	13932.49	3.31	23.07	34.68	66.52
s444	16.99	107	15.88	486.14	13054.5	3.72	23	35.2	65.34
s641	7.37	25.76	28.61	412.46	3709.57	11.12	29.67	42.81	69.31
s713	7.47	29.98	24.92	472.68	5198.38	9.09	27.44	42.59	64.43
s420	37	288.29	12.83	648.38	19023.39	3.41	23.87	41.09	58.09
s386	3.64	21.21	17.16	593.73	5550.74	10.70	19.41	30.39	63.87
s526n	22.25	250.44	8.88	591.57	19034.97	3.11	19.75	43.17	45.75
s526	25.77	210.22	12.26	643.49	18999.31	3.39	20.48	42.38	48.32
s510	6.28	16.78	37.43	944.78	8710.08	10.85	28.66	41.12	69.70
s820	71.77	369	19.45	1596.04	23099.32	6.91	27.11	53.91	50.29
s838	284.64	- ¹	-	871.68	12165.94	7.16	46.66	≥ 44.97	-
s832	106.52	- ¹	-	1706.88	21967.31	7.77	31.16	≥ 54.15	-
s1423	282.11	302.56	93.24	1566.22	13261.03	11.81	51.65	60.86	84.87
s1196	38.33	248.65	15.42	3004.26	22289.32	13.48	30.33	58.54	51.81
s1238	47.45	306.83	15.46	3135.35	29150.62	10.76	31.31	62.47	50.12
s1488	340.09	- ¹	-	4189.69	23664.3	17.70	41.22	≥ 82.16	-
s1494	348.63	- ¹	-	4067.76	22225.33	18.30	42.08	≥ 73.09	-

¹ The naive method failed to terminate before the timeout (6 minutes). In all other runs, the algorithm terminated without timing-out.

TABLE V
USING THE STATE REACHABILITY INFORMATION IN COMBINATIONAL OPTIMIZATION.

Design Being Optimized Benchmark	Design Being Optimized		Optimization 1: Combinational Opt.	Optimization 2: Sequential Optimization ¹	
	Latches	Nodes		Using Approx. Reach	Using Exact Reach.
s27	3	9	7	7	7
s208	8	78	53	53	53
s298	14	106	80	61	61
s344	15	121	98	95	95
s349	15	121	98	95	95
s382	21	140	101	91	91
s400	21	147	101	91	91
s444	21	151	99	89	89
s641	19	172	121	96	96
s713	19	175	123	97	97
s420	16	170	113	113	113
s386	6	169	120	86	86
s526n	21	194	123	85	85
s526	21	195	125	86	86
s510	6	225	207	207	207
s820	5	335	242	237	237
s838	32	358	208	208	-
s832	5	347	247	235	235
s1423	74	466	436	430	-
s1196	18	497	443	443	-
s1238	18	545	447	447	-
s1488	6	674	568	543	543
s1494	6	683	571	551	551
s9234	211	2006	1306	1306	-
s13207	669	3283	2014	1949	-
s15850	597	3843	2670	2529	-
s38417	1636	9696	- ²	- ²	- ²
s38584	1452	13242	9823	9821	-

¹ In each sequential optimization scenario, we optimize starting from the initial design, not from the combinational optimized design.

² The optimization uses MVSIS’s “mfs” command which is BDD-based. The BDDs blew up on these designs.

- The reduction in number of nodes due to purely combinational optimization
- The reduction due to optimization using the approximate reachable state set as EXDCs
- The reduction due to use of the exact set of reachable states used as EXDCs

The experiment was done by running the ABC script “compress2” first to optimize the circuit as much as possible combinational. This script does not make use of don’t cares. Next, we ran MVSIS’s command “mfs” which optimizes the circuit and will use the EXDC network if one is provided. This is the point where state reachability information is used. In the case of the purely combinational optimization, “mfs” was run without an EXDC network. Finally we ran the ABC script “compress2” once more to further optimize. ABC uses as an area metric the number of nodes in the And Inverter Graph (AIG) used for representing the network. The number of nodes in the tables refer to this metric.

We expect the approximate reachability EXDC optimization to be better than purely combinational optimization. We also expect optimization with the exact set of reachable states to be better than optimization with an over-approximation because the exact set presents more EXDC minterms. The surprising result is that in all examples tested, the over-approximation yielded precisely the same optimization as did the exact reachable state set. This means that while in some cases our over-approximation was not so tight, it still contained enough of the useful optimization information.

G. Scaling To Even Larger Designs

In Table VI, we explore an way of scaling our technique to larger designs. Suppose that instead of running our method for 60 seconds and then terminating, we do the following:

```

while (true) {
  Run our method with a timeout of 5 seconds
  Use proved implications to
    (1) Extract node equivalence classes
    (2) Collapse each class into a single node
  Combinationally resynthesize
}

```

In each iteration, we use the partial results obtained after 5 seconds of execution to simplify the network. This allows the next application of the algorithm to proceed further into the design before hitting the time limit. Thus, the design is simplified incrementally, allowing the method to explore ever simpler designs. Additionally, combinational resynthesis changes the nodes in the network, which allows for implications that could not be expressed previously.

In Table VI, we see that this method is extremely powerful. It was applied to the 5 largest designs and resynthesis was iterated 5 times. The reachable state space approximation gets progressively tighter in each iteration, and running for a total of $5 \times 5 = 25$ seconds in this manner gives a much tighter reachability approximation than that provided by the 60 second timeout shown in Table I. This

TABLE VI
REACHABILITY RESYNTHESIS LOOP.

Design	After Iteration:	1	2	3	4	5
s9234: 1345 And Gates ¹ 211 Latches	Approx. Reach. (%), 5 Sec. Timeout	87.47	43.59	36.74	36.69	35.78
	Equiv. Classes Found	0	0	0	0	0
	Average Nodes / Class	0	0	0	0	0
	Gates After Collapsing + Resynthesis ²	1339	1339	1338	1338	1338
s13207: 2092 And Gates ¹ 699 Latches	Approx. Reach. (%), 5 Sec. Timeout	100.00	-	1.48E-4	~ 0.00	~ 0.00
	Equiv. Classes Found	0	6	2	5	2
	Average Nodes / Class	0	5	36.5	15.4	27.5
	Gates After Collapsing + Resynthesis ²	2038	2007	1647	1524	1445
s15850: 2755 And Gates ¹ 597 Latches	Approx. Reach. (%), 5 Sec. Timeout	0.72	0.01	3.50E-3	~ 0.00	~ 0.00
	Equiv. Classes Found	2	2	2	2	2
	Average Nodes / Class	11	13	8	13	11
	Gates After Collapsing + Resynthesis ²	2571	2453	2437	2412	2380
s38417: 8171 And Gates ¹ 1636 Latches	Approx. Reach. (%), 5 Sec. Timeout	99.81	99.81	99.81	99.81	99.81
	Equiv. Classes Found	0	0	0	0	0
	Average Nodes / Class	0	0	0	0	0
	Gates After Collapsing + Resynthesis ²	8117	8117	8117	8117	8117
s38584: 9896 And Gates ¹ 1452 Latches	Approx. Reach. (%), 5 Sec. Timeout	48.39	24.20	5.99	-	0.93
	Equiv. Classes Found	2	2	0	2	2
	Average Nodes / Class	2	2	0	2	5
	Gates After Collapsing + Resynthesis ²	9863	9799	9782	9781	9776

¹ Numbers reported for an AIG after some combinational preprocessing.

² And nodes in an AIG.

preliminary result is interesting but the technique requires further study. Combined with a combinational resynthesis method which uses implications rather than just equivalences, this could yield a powerful sequential resynthesis technique.

V. CONCLUSION

We presented a method to compute over-approximations of the reachable state set. It uses induction to prove implications between nodes in the design that hold in every reachable state. It uses highly tuned data structures and SAT routines, and has the property that it can be terminated at any time but still result in useful data. The longer it is allowed to run, the more implications it will find, and hence the tighter the corresponding over-approximation of the reachable state space. This runtime versus quality trade-off may be what will make this technique useful in practice.

REFERENCES

- [1] M. L. Case and R. K. Brayton and A. Mishchenko, "A Practical Way To Maintain A Transitive Reduction," submitted to *WG, International Workshop on Graph Theoretic Concepts in Computer Science*, 2006.
- [2] C. Hyunwoo and E. Macii and B. Plessier and F. Somenzi and G. Hachtel, "Algorithms for approximate FSM traversal based on state space decomposition," in *DAC*, 1996.
- [3] K. Ravi and K. McMillan and T. Shiple and F. Somenzi, "Approximation and Decomposition of Binary Decision Diagrams," in *DAC*, 1998.
- [4] K. Ravi and F. Somenzi, "High-density reachability analysis," in *ICCAD, Digest of Technical Papers*, Nov. 1995. Pages 154-158
- [5] I.H. Moon and J.Y. Jang and G.D. Hachtel and F.Somenzi and J. Yuan and C. Pixley, "Approximate Reachability Don't Cares for CTL model checking," in *ICCAD, Digest of Technical Papers*, Nov. 1998. Pages 351-358
- [6] J.R. Burch and E.M. Clarke and D.E. Long and K.L. McMillan and D.L.Dill, "Symbolic model checking for sequential circuit verification," in *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, April 1994. Pages 401-424
- [7] K. McMillan, "Don't-care Computation using k-clause Approximation," in *IWLS*, 2005.
- [8] C. van Eijk, "Sequential equivalence checking based on structural similarities," in *IEEE Trans. Computer-Aided Design*, July 2000.
- [9] P. Bjesse and K. Claessen, "SAT-based Verification without State Space Traversal," in *FMCAD*, 2000.
- [10] A.V. Aho and M.R. Garey and J.D. Ullman, "The transitive reduction of a directed graph," in *SIAM J. Comput.*, 1972, Pages 131-137
- [11] K. Simon, "Finding a minimal transitive reduction in a strongly connected digraph within linear time," in *WG '89: Proceedings of the fifteenth international workshop on Graph-theoretic concepts in computer science*, 1990. Pages 245-259
- [12] J.A. La Poutre and J. van Leeuwen, "Maintenance of transitive closures and transitive reductions of graphs," in *Proceedings of the International Workshop WG '87 on Graph-theoretic concepts in computer science*, 1988. Pages 106-120
- [13] J. van Leeuwen, "Graph algorithms," in *Handbook of theoretical computer science, vol. A: Algorithms and Complexity.*, MIT Press, Cambridge, MA, 1990. Pages 525-631
- [14] S. Khuller and B. Raghavachari and N. Young, "Approximating the Minimum Equivalent Digraph," in *SIAM Journal of Computing*, 1995. Pages 859-872
- [15] E. Nuutila and E. Soisalon-Soininen, "On finding the strongly connected components in a directed graph," in *Information Processing Letters*, 1994
- [16] D. Gries and A.J. Martin and J.L. van de Snepscheut and J.T. Udding, "An algorithm for transitive reduction of an acyclic graph," in *Sci. Comput. Program.*, 1989. Pages 151-155
- [17] Berkeley Logic Synthesis and Verification Group, ABC: A System for Sequential Synthesis and Verification, <http://www.eecs.berkeley.edu/alanmi/abc/>
- [18] MVSIS Group. *MVSIS: Multi-Valued Logic Synthesis System*. UC Berkeley. <http://www-cad.eecs.berkeley.edu/mvsis/>
- [19] Niklas Een, Niklas Sorensson, MiniSat. <http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat/Main.html>