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. 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 can find a tight-enough reachable state space approximation to enable circuit optimization, even for very large problems.**

I. Introduction

Given a finite state machine, one is often interested in finding the subset of states that are reachable from a fixed initial state. There are several applications in the fields of synthesis and verification for which knowledge about the reachable state set is invaluable. Some of these 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, and 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 blowup problem which results in failure to terminate on designs of more than 100 or so latches. This approach is not rugged, and this prevents its use in industry. The potential benefits from knowing the reachable state set motivate us to explore a method to do state reachability that is faster and more reliable for large designs.

In this work we propose a fast method to over-approximate the set of reachable states. Given two nodes a and b 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. For an example of this, please see Figure 1.

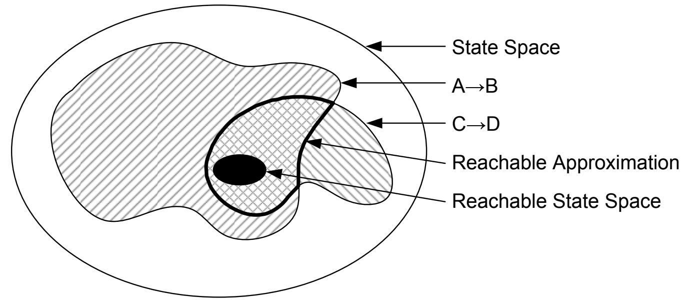

Fig. 1. The implications $A \Rightarrow B$ and $C \Rightarrow D$ each contain the reachable state space, and their product gives us an over-approximation of the reachable state set.

Our approach has the property that every additional implication proved tightens the approximation to the set of reachable states. Therefore the computation can be terminated after proving inductively any new subset of 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 their results are safe for use.

II. Related Work

Several authors have suggested ways to approximate reachability analysis. Most traditional ways simplify BDD-based reachability analysis. Ravi [2] proposed a method to simply the BDD representing the set of reachable states during the progression of the reachability algorithm. Several others [1], [4], [5] have 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-blowup problem. Our method is based on Boolean satisfiability (SAT), and this should allow it to scale better than an optimized BDD-based method.

McMillan [6] presents a novel way to use length-k clauses to approximate don't care sets and accelerate don't care

computation in combinational circuits. Here we find implications between nodes which are essentially 2-literal clauses in sequential circuits. We are specifically interested in finding implications that hold in every reachable state, and this allows us to use a simpler inductive proof technique rather than McMillan's trie quantification approximation.

Van Eijk [7] proposed an inductive technique to discover equivalent nodes between two similar designs. He proposed this as a way to speed up sequential equivalence checking. Bjesse [8] presented a way to improve van Eijk's results by strengthening the inductive hypothesis. In this work we improve on van Eijk's and Bjesse's methods by focusing 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 we provide a framework to trade runtime for quality of results.

III. Finding Implications to Approximate the Reachable State Set

Our method can be most briefly described with 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 we prove candidate implications, how we track implications in the current working set, how we use candidate windowing and timeouts, and how we build the reachable state set over-approximation follow below.

A. Proving a Set of Implications

Suppose we have a set of candidate implications $I = \{(a, b) \mid a \Rightarrow b\}$. We use an inductive technique called k-step induction to prove that a subset of I holds in every reachable state.

In the base case we examine all states reachable in k or less transitions from the initial state. We keep only those elements of I which hold in 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. Again, we discard all members of I which fail to hold in this context.

At the end of the procedure, we have refined I to exactly those implications which hold in every reachable state, and we have proved this by induction rather than reachability analysis.

As an 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 nodes in the network.

We unroll the circuit $k = 2$ times by adjoining 2 consecutive transition relations. 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 of the proof step involves checking that the following expression holds in the initial state and for all 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 $(d \Rightarrow e)_2$ does not hold. This says that there is a state reachable in one step from the initial state for which this implication does not hold. Remove $d \Rightarrow e$ from the candidate set of implications.

In the inductive step, we make no assumptions about the state of the first frame, i.e. we let all latches and primary inputs be free variables. We then check

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

Suppose we find a set of inputs such that the left hand side holds, and $(c \Rightarrow d)_2$ does not hold. The left hand side characterizes our current approximation to the set of reachable states. The above then says that in a state that we currently estimate to be reachable, there is a next state where $c \Rightarrow d$ does not hold. Therefore $c \Rightarrow d$ does not hold in every reachable state, and we remove it from our candidate set of implications.

If no other implications are disproved then we know that $\{a \Rightarrow b, b \Rightarrow c, c \Rightarrow d\}$ hold in every reachable state.

B. Window Proof Technique

We use 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.

The set of candidate implications is pruned until every implication in the set holds in all reachable states. Until this fixed point is reached, we cannot say that the conjunction of the implications gives an over-approximation to the reachable state set. It is desirable to be able to terminate early with useful partial results, and so we use an ever widening window. At each step, the candidate implications in the window are pruned until a fixed point is reached for that window. Then the window is widened and a proof of a new set of implications begins anew. 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, and this helps to speed up the next proof by reducing its estimate of the reachable state set and tightening the left hand side of the inductive hypothesis. Implications which fail to hold in a particular window must be considered as candidates in the next window iteration since we may learn more about the reachable state space in the next window, and this new information may cause a previously false implication to now appear true.

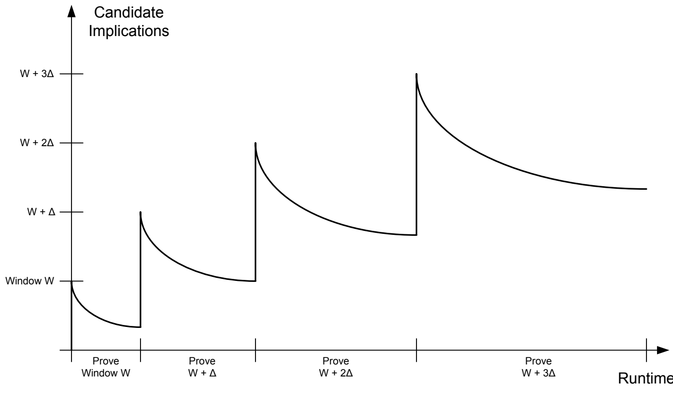

Fig. 2. Our hierarchical proof technique.

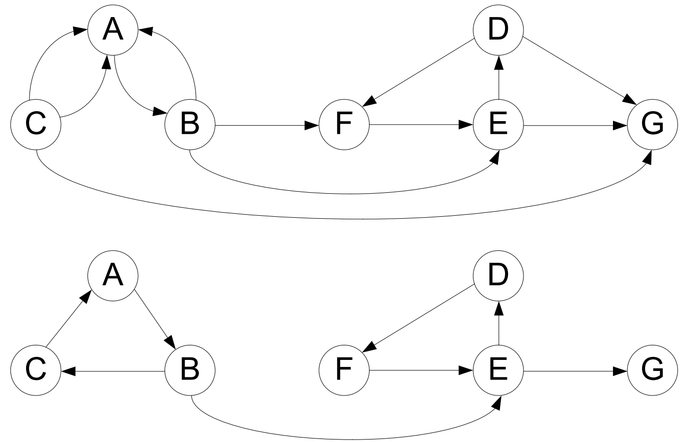

Fig. 3. An example directed graph and its transitive reduction.

For an example of this technique, see Figure 2. Here we start with an initial window W and a set of candidate implications. We prune this set until we reach a fixed point, and then we widen the window to $W + \Delta$. This increases the candidate implication set, and we must again apply our inductive proof technique to prune the candidate set to only those implications that hold in all reachable states. If we are interrupted at any time during this proof of the $W + \Delta$ implications, we can return the fixed point reached using only the W candidates.

An open problem is how to best choose the candidate implication window. This depends heavily on the target application for which reachability analysis is being applied. For reachability used in conjunction with combinational optimization, a random candidate window seems to work best. For property verification, we want to show that a set of bad states is unreachable. In this 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 we are successful in proving an implication the state space over-approximation, the conjunction of the implications (Section III-D), will evaluate to 0 on a set of bad states. This tells us that these bad states are unreachable.

C. The Implication Graph

Throughout our algorithm, we must maintain the set of candidate implications. These implications can be thought of as forming a directed graph. The nodes of the graph are the nodes of the original logic network, and each edge represents one implication.

There is a graph theoretic procedure that can minimize our implication graph. This minimization allows us to dramatically compress our candidate set of implications. In practice, we observe that by paying the overhead to compress the graph, the candidate implication set can be reduced to less than 5% of its original size. This dramatically speeds up our overall algorithm.

The key property is transitivity. If $a \Rightarrow b$ and $b \Rightarrow c$ then necessarily $a \Rightarrow c$. 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 some alternate path. Aho et. al. [9] 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 then $r(G)$ 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 known as the *condensed graph*. It is acyclic and can be reduced uniquely. Therefore the transitive reduction of a general directed graph is unique in cardinality.

An example of transitive reduction is shown in Figure 3. 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 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. We must avoid adding redundant edges. Also, reducing paths for some pre-existing edges

may be introduced. These must be efficiently identified and removed.

Edge deletion. When we refine our candidate implication set, we remove implications that we know don't hold in every reachable state. This requires us to remove an edge from the implication graph. There are in general a number of edges not represented in the transitive reduction because they are redundant, but removing a single edge may make the edges now irredundant. These edges 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, if we remove $b \Rightarrow c$ is removed then $a \Rightarrow c$ must be explicitly added back into the graph. In this way removing an edge from a transitive reduction may increase the size of the graph.

The graph algorithms implemented in 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.

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:

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

Because every implication holds in all reachable states, we may safely universally quantify out all primary inputs from every implication to get an equivalent expression that holds in every reachable state. Taking the conjunction of expressions that hold in reachable states yields a product that also holds in every reachable state, and this product 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 it is only required if one wants to express the over-approximation in terms of the latches. Another way is to simply assert the proved implications as additional SAT clauses an application. Hence, 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 [16] logic optimization and verification framework, and utilizes the SAT solver MiniSat 1.14 [18].

Table I compares the performance of our algorithm against a BDD-based reachability algorithm implemented in MVSIS [17]. The ISCAS89 benchmarks were run through both systems on a 3 GHz Pentium 4 machine. The BDD method was given a time limit of 1 hour, and our method was given a time limit of 2 minutes. The last two columns in the table show the percentage of states that are estimated to be reachable, as a percentage of the total state space (2^n for n latches). When our method reports that more states are reachable than the exact method reports, then the difference is the reachable state space over-approximation. It is interesting to note that our approximation is not always tight, but it does manage 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. Our method is much more rugged than the BDD based method in the sense that it is able to process very large designs, and it is able to find results quickly.

Table II illustrates the power of 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 satisfy a conjunction of implications. It is not obvious that such a conjunction could ever completely characterize the reachable state set. Furthermore, we only consider implications between nodes pre-existing in the design. A design may lack nodes conducive to implications, and this may 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 we have an expression R such that $R = 1$ iff the current state is reachable then we can project R onto the set expressible with implications by finding:

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

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

Table II also shows the effect of increasing the parameter k in "k-step induction" (Section III-A). There is a danger in the k-step induction that the SAT solver may find an unreachable state U such that:

- A particular implication holds in a sequence of k unreachable states starting from U.
- The implication fails to hold in a next state reachable from the k'th state.
- The implication is otherwise true.

Thus the inductive hypothesis can spuriously disprove implications that may hold in every reachable state. Increasing the value of k strengthens the inductive hypothesis by decreasing the probability that such bad states exists. Therefore we expect our results to get better as k increases. In Table II we show the results for increasing k. In most cases the over-approximation gets tighter, and the over-approximation is always lower bounded by the projection of the exact reachable state set onto the implication space.

Table II also illustrates the disadvantage of increasing k. The size of the SAT problem increases as k increases, and

TABLE I

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

Benchmark	Circuit Statistics[1] Levels	Latches	Nodes	Runtime (sec) Exact Reach.	Approx. Reach.	Reachable States (%) Exact Reach.	Approx. Reach.
s27.blif	5	3	8	0.07	0.40	75.00	75.00
s208.blif	9	8	71	0.10	1.59	100.00	100.00
s298.blif	9	14	100	0.13	2.21	1.33	3.10
s344.blif	13	15	104	0.18	1.34	8.01	70.73
s349.blif	13	15	104	0.20	1.57	8.01	70.73
s382.blif	12	21	132	0.14	56.93	0.42	6.71
s400.blif	13	21	139	0.18	62.51	0.42	6.71
s444.blif	14	21	143	0.22	72.38	0.42	6.66
s641.blif	25	19	146	0.39	21.70	0.29	0.42
s713.blif	25	19	149	0.44	29.20	0.29	0.42
s420.blif	11	16	157	1.76	125.06	100.00	100.00
s386.blif	10	6	166	0.08	35.69	20.31	20.31
s526n.blif	9	21	185	0.24	121.32	0.42	5.88
s526.blif	9	21	186	0.17	121.45	0.42	5.50
s510.blif	11	6	213	0.11	10.37	73.44	73.44
s820.blif	14	5	331	0.12	129.17	78.12	78.12
s838.blif	15	32	333	3599.51	127.97	-	100.00
s832.blif	14	5	343	0.10	130.71	78.12	78.12
s1423.blif	54	74	459	3605.45	129.16	-	37.61
s1196.blif	19	18	477	0.37	49.53	1.00	56.84
s1238.blif	21	18	525	0.39	67.29	1.00	56.84
s1488.blif	15	6	662	0.13	130.98	75.00	75.00
s1494.blif	15	6	671	0.13	132.77	75.00	75.00
s5378.blif	17	164	1299	3599.65	128.67	-	- [2]
s9234.blif	32	211	1791	3599.75	130.58	-	70.19
s13207.1.blif	33	638	2621	3599.66	141.88	-	- [2]
s13207.blif	34	669	2621	3599.82	131.91	-	18.76
s15850.blif	45	597	3357	3599.63	130.52	-	0.31
s38417.blif	31	1636	9063	3599.70	142.85	-	99.90
s38584.blif	36	1452	11644	3599.76	143.06	-	99.99

[1] Numbers reported after some combinational preprocessing.

[2] 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 II

COMPARING k-STEP INDUCTION RESULTS TO THE IDEAL IMPLICATION RESULTS

Benchmark	Circuit Statistics Levels	Latches	Nodes	Exact Reachability Projection [1] Exact Reach.	Projected to Imps.	Approximate Reachability, Sweeping k [12] $k=1$	$k=2$	$k=3$
s27.blif	5	3	8	75.00	75.00	75.00	75.00	75.00
s208.blif	9	8	71	100.00	100.00	100.00	100.00	100.00
s298.blif	9	14	100	1.33	3.05	3.10	3.10	3.10
s344.blif	13	15	104	8.01	70.53	70.73	70.73	70.73
s349.blif	13	15	104	8.01	70.53	70.73	70.73	70.73
s382.blif	12	21	132	0.42	1.47	6.71	6.71	6.71
s400.blif	13	21	139	0.42	1.47	6.71	6.71	6.71
s444.blif	14	21	143	0.42	1.51	6.66	6.66	6.66
s641.blif	25	19	146	0.29	0.42	0.42	0.42	0.42
s713.blif	25	19	149	0.29	0.47	0.42	0.42	0.42
s420.blif	11	16	157	100.00	100.00	100.00	100.00	100.00
s386.blif	10	6	166	20.31	20.31	20.31	20.31	20.31
s526n.blif	9	21	185	0.42	1.44	5.88	6.62	9.08
s526.blif	9	21	186	0.42	1.44	5.50	5.51	5.51
s510.blif	11	6	213	73.44	73.44	73.44	73.44	73.44
s820.blif	14	5	331	78.12	78.12	78.12	81.25	81.25
s832.blif	14	5	343	78.12	78.12	78.12	78.12	78.12
s1423.blif	54	74	459	-	-	37.61	37.13	26.48
s1196.blif	19	18	477	1.00	15.62	56.84	56.84	56.84
s1238.blif	21	18	525	1.00	15.62	56.84	56.84	56.84
s1488.blif	15	6	662	75.00	75.00	75.00	75.00	75.00
s1494.blif	15	6	671	75.00	75.00	75.00	75.00	75.00
s9234.blif	32	211	1791	-	-	70.19	17.54	7.57
s13207.blif	34	669	2621	-	-	18.76	0.00	0.00
s15850.blif	45	597	3357	-	-	0.31	0.34	0.13
s38417.blif	31	1636	9063	-	-	99.90	71.38	71.88
s38584.blif	36	1452	11644	-	-	99.99	43.75	99.99

[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.

so the algorithm slows down. Under a time limit this can cause less implications to be proved, and may actually degrade the performance. We observe that the benefit obtained by increasing k above 1 is negligible, and so for all results other than Table II we fix $k = 1$.

In Table III, we varied the maximum processing time allowed in our implementation. As we increase the processing time 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 this principle in action. Our method has a runtime versus quality trade-off and in this way gives the user more control.

In the introduction we mention that the reachable state information can be used by a combinational optimizer by feeding the optimizer the unreachable states as external don't cares (EXDCs). In Table IV, 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 obtained by running our algorithm for 2 minutes with $k = 1$. Table IV shows three meaningful figures:

- the reduction in number of nodes due to purely combinational optimization
- the reduction in the number of nodes 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 optimization was done by first running the ABC script "compress2" to optimize the circuit as much as possible combinationally. This script does not make use of don't cares. Next, we ran MVSIS's command "mfs" which optimizes the circuit using an EXDC network. This is the point where the state reachability information is used. Finally we ran the ABC script "compress2" once more to further optimize after the EXDC optimization. ABC uses as an area metric, the number of nodes in the and inverter graph 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 fact to note here is that in all designs benchmarked, the reachable state space over-approximation yielded precisely the same optimization as did the exact reachable state space. This means that while in some cases our over-approximation was not so tight, it contained enough information to enable logic optimization.

V. Conclusion

We have presented a method to compute an over-approximation of the reachable state set. The method works by inductively proving that implications between nodes in the design hold in every reachable state. It does this using highly tuned data structures and SAT routines. It has the property that the longer it is allowed to run, the more implications it will find. The corresponding over-approximation of the reachable state space will then be tighter. This runtime versus quality trade-off is what makes may make this technique useful in practice.

We believe this method presents a promising alternative to BDD-based explicit state reachability.

References

[1] 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.

[2] K. Ravi and K. McMillan and T. Shiple and F. Somenzi, "Approximation and Decomposition of Binary Decision Diagrams," in *DAC*, 1998.

[3] K. Ravi and F. Somenzi, "High-density reachability analysis," in *ICCAD, Digest of Technical Papers*, Nov. 1995. Pages 154-158

[4] 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

[5] 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

[6] K. McMillan, "Don't-care Computation using k-clause Approximation," in *IWLS*, 2005.

[7] C. van Eijk, "Sequential equivalence checking based on structural similarities," in *IEEE Trans. Computer-Aided Design*, July 2000.

[8] P. Bjesse and K. Claessen, "SAT-based Verification without State Space Traversal," in *FMCAD*, 2000.

[9] 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

[10] 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

[11] 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

[12] J. van Leeuwen, "Graph algorithms," in *Handbook of theoretical computer science, vol. A: Algorithms and Complexity.*, MIT Press, Cambridge, MA, 1990. Pages 525-631

[13] S. Khuller and B. Raghavachari and N. Young, "Approximating the Minimum Equivalent Digraph," in *SIAM Journal of Computing*, 1995. Pages 859-872

[14] E. Nuutila and E. Soisalon-Soininen, "On finding the strongly connected components in a directed graph," in *Information Processing Letters*, 1994

[15] 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

[16] Berkeley Logic Synthesis and Verification Group, ABC: A System for Sequential Synthesis and Verification, http://www.eecs.berkeley.edu/ alanmi/abc/

[17] MVSIS Group. *MVSIS: Multi-Valued Logic Synthesis System*. UC Berkeley. http://www-cad.eecs.berkeley.edu/mvsis/

[18] Niklas Een, Niklas Sorensson, MiniSat. http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat/Main.html

TABLE III

COMPARISON OF THE QUALITY OF RESULTS OVER TIME.

Benchmark	Circuit Statistics Levels	Latches	Nodes	% Reachable After a Fixed Amount of Processing Time 20 sec	40 sec	60 sec	80 sec	100 sec	120 sec
s27.blif	5	3	8	75.00	75.00	75.00	75.00	75.00	75.00
s208.blif	9	8	71	100.00	100.00	100.00	100.00	100.00	100.00
s298.blif	9	14	100	3.10	3.10	3.10	3.10	3.10	3.10
s344.blif	13	15	104	70.73	70.73	70.73	70.73	70.73	70.73
s349.blif	13	15	104	70.73	70.73	70.73	70.73	70.73	70.73
s382.blif	12	21	132	15.87	6.73	6.71	6.71	6.71	6.71
s400.blif	13	21	139	6.71	6.71	6.71	6.71	6.71	6.71
s444.blif	14	21	143	14.00	6.96	6.96	6.66	6.66	6.66
s641.blif	25	19	146	0.42	0.42	0.42	0.42	0.42	0.42
s713.blif	25	19	149	0.42	0.42	0.42	0.42	0.42	0.42
s420.blif	11	16	157	100.00	100.00	100.00	100.00	100.00	100.00
s386.blif	10	6	166	20.31	20.31	20.31	20.31	20.31	20.31
s526n.blif	9	21	185	21.42	23.46	9.08	9.08	5.88	5.88
s526.blif	9	21	186	24.17	20.75	9.34	9.34	5.50	5.50
s510.blif	11	6	213	73.44	73.44	73.44	73.44	73.44	73.44
s820.blif	14	5	331	81.25	81.25	81.25	81.25	78.12	78.12
s838.blif	15	32	333	100.00	100.00	100.00	100.00	100.00	100.00
s832.blif	14	5	343	84.38	81.25	81.25	78.12	78.12	78.12
s1423.blif	54	74	459	32.58	35.88	40.09	37.13	37.61	37.61
s1196.blif	19	18	477	56.84	56.84	56.84	56.84	56.84	56.84
s1238.blif	21	18	525	58.79	56.84	56.84	56.84	56.84	56.84
s1488.blif	15	6	662	75.00	75.00	75.00	75.00	75.00	75.00
s1494.blif	15	6	671	85.94	75.00	75.00	75.00	75.00	75.00
s9234.blif	32	211	1791	95.91	99.98	99.98	70.15	70.15	70.19
s13207.1.blif	33	638	2621	99.26	97.83	98.65	98.65	98.46	- [1]
s13207.blif	34	669	2621	2.01	0.58	0.24	18.76	18.76	18.76
s15850.blif	45	597	3357	6.96	2.88	2.88	0.29	0.29	0.31
s38417.blif	31	1636	9063	100.00	100.00	78.09	100.00	78.09	99.90
s38584.blif	36	1452	11644	50.00	50.00	50.00	50.00	50.00	99.99

TABLE IV

USING THE STATE REACHABILITY INFORMATION IN COMBINATIONAL OPTIMIZATION.

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

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

[2] The optimization uses MVSIS's "mfs" command which is BDD-based. The BDDs blew up on these designs.