m-Inductive Properties of Logic Circuits

Hamid Savoj, Alan Mishchenko, and Robert Brayton
University of California, Berkeley

Abstract—**This paper introduces the concept of *m*-inductiveness over a set of signals *S* in a sequential circuit. The *m*-inductive property can be used for equivalence-checking or improved sequential optimization. When applied to verification, only equivalency of signals in *S* and primary outputs need to be checked. When applied to optimization, it allows the behavior of many next-state functions (not in *S*) to be changed while maintaining correctness at the primary outputs of a circuit, creating flexibility that can be used for sequential optimization. It is shown that the number of signals in *S* is reduced as *m*, the parameter for the induction, increases. We provide an algorithm for finding a minimal set *S*, as well as one for using *m*-inductiveness in optimization. We also give examples of industrial designs where *m*-inductive based optimization results in significant area reduction.**

I. Introduction

A sequential circuit, depicted in Figure 1, is composed of a combinational logic part (A^1) and a set of memory elements (called flip-flops (FF), or flops), a set of primary inputs (*PI*) and a set of primary outputs (*PO*). The combinational logic A^1 is a Boolean network which has inputs composed of the *PIs* plus the current states (*CS*) of the FFs, and outputs composed of the *POs* plus the next-state functions (*NS*). The values of the *NS* functions are stored in the FFs on the next clock edge.

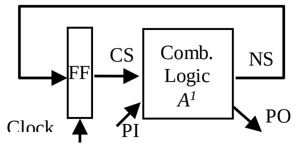


Figure 1: A Sequential Circuit *A*

We assume that the FFs start in a known initial state. Two sequential circuits, *A* and *B*, are said to be equivalent (denoted *A*=*B*) if, starting from their respective initial states, for any given (infinite) sequence of *PIs*, the generated (infinite) sequences of the two associated *POs* are identical.

If only combinational don't cares are used for optimization to convert a circuit A^1 to B^1, sequential equivalence checking (*A*=*B*) consists of just checking the equivalence of their combinational parts ($A^1 = B^1$). In contrast, sequential synthesis may use information that can change the behavior and number of NS functions but maintain the behavior observed at the POs. This information can lead to (a) using the unreachable states as don't cares, (b) using the equivalence of two states, (c) using a different state-encoding for the reachable states, (d) retiming the FF's, or (e) merging two signals if they have the same values in all reachable states.

In this paper, we introduce the concept of *m-inductiveness* for a given sequential machine. It can be used to verify two sequential machines are equal, to convert the sequential equivalence problem into a combinational one, to compute larger don't care sets for optimization, and to change the functions of many FFs while preserving the behavior at the POs. The concept is related to an inductive proof except there is no base case, only the inductive case. Roughly we consider two copies of the same sequential machine and look for a set of signals in one copy, which when constrained for *m* cycles to be equal to their twins in the other copy, can be proved equal in the following cycle (*m+1)*. Typically the signals are the POs and a subset, *S*, of the FFs. This concept generalizes the results in [14] which required that all next-state functions be in *S*. Since the signals not in *S* need not be restricted, they can be used as additional flexibility in synthesis or in checking equivalence after sequential synthesis. The latter was the main application in [14], e.g. sequential equivalence checking after clock-gating, however in this paper, we apply these ideas to sequential synthesis.

The organization of this paper is as follows. Section II reviews methods for using don't cares as they arise and are used in synthesizing sequential machines. Section III defines *m*-inductiveness, how it can be used for equivalence checking, how it can result in larger sequential observability don't cares, and in optimizing pipelines. Section IV provides an algorithm for finding a minimal set of nodes for a circuit to be *m-inductive*. Section V shows two sets of experimental results applied to ISCAS and industrial designs. In the first, we apply the algorithm of Section V to find a minimal set *S* of FFs for *m*=1, and discuss the resulting sizes of *S* relative to the number of FFs. In the second, we use the additional flexibility provided by *S* for sequential synthesis to derive smaller sequential machines.

II. Don't care methods for Sequential synthesis

A. *Combinational Don't Care*

Combinational synthesis changes only the combinational circuit A^1 of a sequential machine but keeps the *POs* and *NS* functions, as functions of *PIs* and *CS* variables, unchanged. The Boolean network A^1 can be completely restructured as a result of combinational synthesis. Don't care conditions in A^1 can be used during synthesis to find a good implementation of A^1. Such don't care conditions are satisfiability don't cares (SDCs), observability don't cares (ODCs), and external don't cares (EDCs). SDCs and ODCs are related to the structure of the network A^1 while EDCs are either computed from sequential behavior of the circuit (e.g. unreachable states), extracted from a hierarchy (if a design is hierarchical), or supplied by the user. The *NS* functions and *POs* can change if external don't cares are used during optimization.

Definition 1: *A Boolean Network A^1 is a directed acyclic graph (DAG) such that for each node in A^1 there is an associated representation of a Boolean function f_i, and a Boolean variable y_i, where $y_i = f_i$. There is a directed edge (i, j) from y_i to y_j if f_j explicitly depends on y_i or $\overline{y}_i$. The global function g_i at y_i is the function at the node expressed in terms of PIs and CS variables.*

Definition 2: *If y_i is the variable at an intermediate node of A^1 and f_i its logic function, then $y_i = f_i$; therefore, we don't care if $y_i \neq f_i$ (i.e. $y_i \oplus f_i$ is don't care). The expression $SDC = \sum_{i=1}^{m}(y_i \oplus f_i)$ is the satisfiability don't care set of A^1.*

Simulation of A^1 with each of *PI* and *CS* minterms in the Boolean space B^n forces the value of each intermediate node to 0 or 1. Some combinations of values on the intermediate variables are not possible. The SDC of A^1 contains all the impossible combinations in the Boolean space B^{n+m}.

Definition 3: *The observability don't cares (ODCs) at each intermediate node y_i of a multi-level network are input minterms under which y_i can toggle between 0 and 1 without causing any of the POs and NS functions to toggle.*

The ODC of an intermediate node y_i can also be expressed in terms of intermediate nodes of the network. For example, the ODC at node y_2 in Figure 2 is $ODC_2 = y_1y_3 + \bar{y}_1\bar{y}_3$. This means that whenever both y_1 and y_3 are both 1 or both 0, the value of y_2 has no effect at any of the outputs. The same ODC in terms of inputs is $ODC_2 = x_1x_2 + \bar{x}_1\bar{x}_2 + \bar{x}_3$. The ability to express the ODCs in terms of various sets of intermediate variables is important to node optimization. Once we have ODCs at node y_2, we can set y_2 to 0.

An interesting fact is that the ODC at a node can intersect the SDC of the network. In Figure 2, the cube $y_1\ y_2\ y_3$ is in ODC and also in the SDC because y_1 and y_2 cannot be equal to 1 at the same time under any input combination.

Definition 4: *The external don't care (EDC) set for each PO of A^1 captures all input minterms for which the value of that output is not important.*

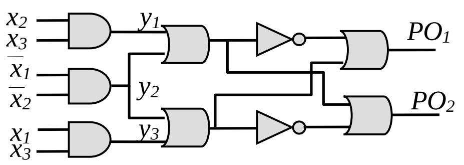


Figure 2: ODCs intersect SDC

EDCs can come from many sources. If circuit A^1 is cut out of a larger circuit as shown in Figure 3 (e.g. a module in a hierarchical design), some input minterms may not be possible because of SDCs of the larger circuit. In this case, inputs of A^1 are intermediate nodes of the larger circuit and they may not produce all combinations of minterms. The EDCs can come from the ODCs of the larger circuit. Each PO of A^1 is an intermediate node in the larger circuit and a change in that PO may not be visible in the outputs of the larger circuit for certain input minterms of A^1. If EDCs of A^1 come from the ODCs of intermediate nodes of a larger circuit, they may not be *compatible*.

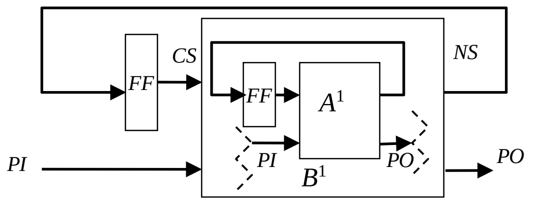


Figure 3: Extracting EDCs

B. Don't Care Methods for Sequential Synthesis

Our discussions use the following notation:

1. *B^r denotes the full Boolean space of the r state variables in a machine A*
2. *T_0^A denotes a subset of B^r. Denote T_n^A as the image of T_0^A after n cycles of machine A, i.e. $s \in T_n^A \Leftrightarrow \exists s_0 \in T_0^A,\ s_0 \overset{A^n}{\Rightarrow} s$.*
3. *$[A = B]_S$ denotes that machine A is equivalent to B for all initial states $s \in S$.*

The unreachable states of a machine *A* is the set of states that are not reachable starting from its initial state. Since such states are never reached, they can be used as don't cares to optimize A^1 into B^1, and hence into an equivalent sequential machine *B*. They can be viewed as satisfiability don't cares, SDCs.

One way to compute the unreachable state set is to start from the initial state ($\{s_0\} \equiv T_0^A$) and recursively compute the images $\{T_i^A\}$ under *A* until no new states are reached, i.e. $\sum_{i=0}^{n+1} T_i^A = \sum_{i=0}^{n} T_i^A$. However, the number of iterations *n* needed to achieve this fixed point and the size of the reached state set can be exponentially large, so this method is rarely used in practice.

Another approach looks for an "invariant" $T_0^A \subset B^r$ where $T_1^A \subseteq T_0^A$. The condition $T_1^A \subseteq T_0^A$ is enough to guarantee that $T_{k+1}^A \subseteq T_k^A, \forall k \geq 0$.

Lemma 1: *If $T_1^A \subseteq T_0^A$, then $T_{k+1}^A \subseteq T_k^A, \forall k \geq 0$.*

Proof: Clearly, the base case holds, $T_1^A \subseteq T_0^A$. Assume $T_k^A \subseteq T_{k-1}^A$, but $T_{k+1}^A \not\subseteq T_k^A$. Then $\exists s \in T_k^A$ whose image $s' \notin T_k^A$. But $s \in T_{k-1}^A$ and therefore $s' \in T_k^A$, a contradiction. Therefore, by induction, $T_{k+1}^A \subseteq T_k^A, \forall k \leq 0$. **QED**

Therefore, if $s_0 \in T_0^A$ and $T_1^A \subseteq T_0^A$, then T_0^A is superset of the reachable states and its complement can be used as don't cares.

If one starts with $T_0^A = B^r$, Lemma 1 clearly holds and $T_{k+1}^A \subseteq T_k^A, \forall k \geq 0$. As long as T_n^A includes the initial state, it is a superset of the reachable states and can be used to optimize *A* to a new machine *B*. One way to achieve this optimization is to build the unrolling A^{n+1}, and use the first *n* copies of A^1 to optimize the last copy of A^1 as shown in Figure 4. The SDCs at the current state inputs of the last copy of A^1 is the complement of T_n^A. But, it is still necessary that $s_0 \in T_n^A$.

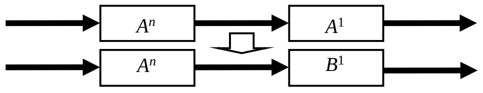


Figure 4: Optimization using SDCs from A^n to simplify A^1 resulting in B^1.

A known approach for computing a superset of reachable states is called signal correspondence. It effectively computes such a set by *k*-induction. Machine *A* is simulated for many clock cycles stating from its initial state. Signals with equal, complementary, or constant values throughout the simulations are identified and postulated to be always in the same relation on the set of reachable states. These are checked for the first *k* steps as the base case for induction, and for the inductive case, they are assumed to hold for *k* cycles. Then an attempt is made to prove that all hold on the next cycle. If not all can be proved, the set is trimmed to only those that could be proved. This is iterated until a fixed point is found. The inductive step can be done efficiently using SAT, hence this method is practical for large machines. Note that the superset of reachable states is given implicitly as the set of states on which the equalities hold. Optimization of the sequential circuit is done by merging the signals proven equal and setting the constant signals to their values and propagating the constants.

A more recent method, discussed in [14], is to optimize a sequential circuit, *A*, using a *k*-step unrolling process as indicated in Figure 5. Let $C_1 = A^k$ denote this combinational circuit, which has *k* copies of A^1. The outputs of C_1 are the *k* PO sets of *A*, one set for each time-frame plus the final *NS* signals at the output of the k^{th} frame. The inputs of C_1 are the *k* PI sets, one for each frame, plus the initial *CS* signals of the first frame.

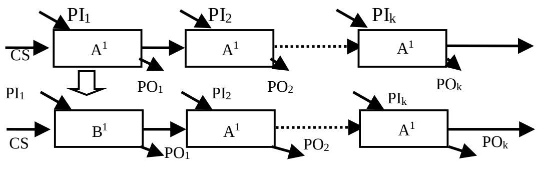


Figure 5: *k*-step unrolling, ODC optimization of A^1

Then *combinational* synthesis is applied to the logic of the first copy of A^1 and the other *k*-1 copies are left untouched, resulting in the combinational circuit $C_2 = B^1A^{k\text{-}1}$ (bottom of Figure 5). Although the global functions at the *NS* outputs of the *final* copy are necessarily preserved by the combinational synthesis, the internal *NS* outputs of B^1 as well as the successive *NS* outputs of the *k*-2 copies of A^1 in C_2 may be functionally different from the corresponding internal *NS* outputs in C_1.

The purpose of the last *k*-1 copies of A^1 in this scenario is to produce Observability Don't Cares (ODCs) used for transforming the first copy of A^1 into B^1, which will form the combinational part of a new sequential circuit *B*. This sequential circuit is shown in Figure 6.

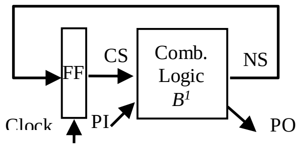


Figure 6: Derived Sequential Circuit *B*

Since the changes in B^1 are not observable at any of outputs of C_1, $B^1A^{k-1} = A^k$. It is proved in [14] that this combinational equivalence implies the sequential equivalence $[A = B]$, for any initial state s_0.

III. *M*-INDUCTIVE PROPERTIES OF SEQUENTIAL CIRCUITS

This section extends and generalizes the work in [14] and shows techniques to further enlarge sequential ODCs. In [14], the synthesis from A^1A^{k-1} to B^1A^{k-1} required that all outputs (POs and final NS outputs) of B^1A^{k-1} be equal to the corresponding outputs A^k. The motivation of this generalization is the following. In many circuits preserving only a subset of *NS* outputs plus all POs is sufficient to have *A* = *B*. Optionally, some of the *NS* outputs that need to be observed at the final copy of A^1 can be replaced by internal nodes in synthesizing A^k into B^1A^{k-1}. Thus, there is no need to preserve all *NS* outputs during optimization. Since only a subset of *NS* outputs needs to be preserved, the ODCs computed at the first copy of A^1 in A^k are larger.

An extension, given by Theorem 3 below, is to maintain equality between a subset of intermediate nodes and *NS* outputs in the last *m* copies of A^1 in B^1A^{k-1} and those of A^k, instead of maintaining all FFs in the last copy A^1. This can reduce the number of unique *NS* outputs or intermediate nodes that must be maintained, possibly further increasing the DCs that can be used in synthesis.

A. Definition of m-Inductive Property

To reduce the number of observed *NS* outputs, machine *A* needs to have a special *m-inductive* property relative to a set of signals *S*. Figure 7 shows two copies of *A* forming circuits A_x and A_y. The two circuits differ only in the initial states, *x* and *y*, in their FFs. Thus the POs of A_x and A_y are not necessarily equal. *S* is a subset of intermediate nodes and/or *NS* outputs of A^1 as shown in Figure 7.

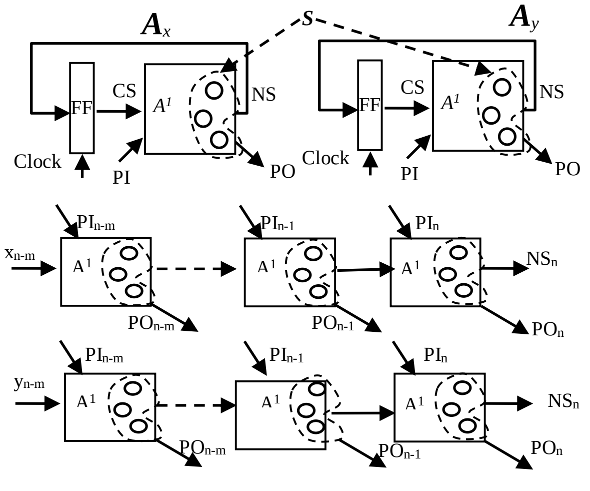


Figure 7: *m+1* unrolling of A_X and A_Y.

Definition 5: *Given a machine A, let A_x and A_y denote identical machines, except for their initial states, x and y. Let S be a set of nodes, which may include intermediate as well as NS nodes of A. A has the* <u>*m*-inductive property under S</u>*, if the equality of the corresponding nodes S and POs in A_x and A_y at* *times n-m to n-1 imply those nodes and POs are also equal at time n.*

Figure 7 shows two *m*+1 step unrolling's of *A* for times *n-m* to *n*. The current states x_{n-m} and y_{n-m} at time *n-m* may be different. It is assumed that the corresponding nodes in the set *S* and the POs in both unrolling's are equal at times *n-m to n-1*. The *m*-inductive property of *A* under *S* implies that at time *n* the corresponding nodes in *S* and POs are also equal in both circuits. Informally, we can view this property as similar to the inductive step in *m*-step induction used in signal correspondence; however, the equalities used are between signals of two versions of the same circuit.

An interesting sufficient condition for *m*-inductiveness under *S* is stated below.

Theorem 1: *Machine A is m-inductive under set S if the nodes in S and POs at time n only depend on a) the values of the nodes in S and POs at times n-m to n-1, and b) the PIs at times n-m to n.*

Proof: Suppose the hypothesis holds, Then if the corresponding nodes in *S* and outputs are equal from times *n-m* to *n*-1 for two copies A_x and A_y of Machine *A*, those nodes are also equal at time *n* since by assumption the function of those nodes depends only on the values of nodes in *S* and outputs from time *n-m* to *n*-1 and inputs from time *n-m* to *n*. Therefore, *A* is *m*-inductive under *S*. **QED**

Example: Figure 8 shows a circuit where equality of the output *q* at time *n*-1 for two copies of the machine, A_x and A_y, implies the equality of *q* at time *n*. This is because q_n can be expressed in terms of inputs and q_{n-1}, and there is no other dependency on the current state variables:

$$q_n = r_{n-1}\left(e_{n-1}s_{n-1}t_{n-1} + \bar{e}_{n-1}CS1_{n-1}CS2_{n-1}\right) = r_{n-1}\left(e_{n-1}s_{n-1}t_{n-1} + \bar{e}_{n-1}\, q_{n-1}\right)$$

In this case, the set *S* is empty; no *NS* outputs or intermediate nodes are needed in *S*. This machine is 1-*inductive* under the empty set *S*.

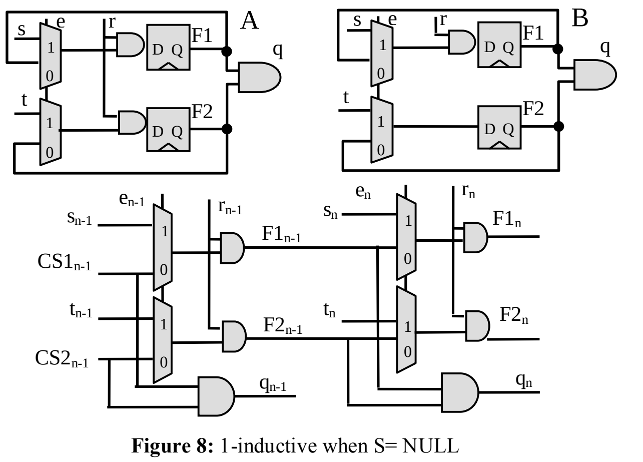


Figure 8: 1-inductive when S= NULL

Lemma 2: *Machine A is* 1-*inductive under S if S includes all NS outputs of the machine A.*

Proof: In this case, if the two copies, A_x and A_y, are equal at time *n*, they will be equal at time *n*+1 since the CS variables and inputs going to both machines at time *n*+1 are the same.

B. Equivalence of m-Inductive Circuits

Theorem 2: *Given machines A and B and corresponding nodes S in both machines. If all corresponding nodes in S and POs of the two machines are equal for the first m cycles starting from the initial states, and both machines are m-inductive under S, then the two machines are sequentially equivalent.*

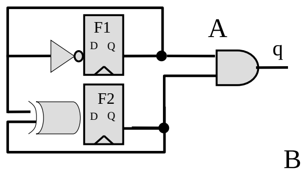
Proof: The two machines are equal for the first *m* cycles starting from the initial states. Also, equality of corresponding nodes in *S* and POs for *m* cycles produces equal nodes in *S* and outputs for both machines on the next cycle. Therefore, the machines are equal by induction. **QED**

Example: Figure 9 shows two implementations of a counter that produces the sequence *000100010001...* at the output *q*. Machine *A* is a two-bit counter initialized at *F*1=0 and *F*2=0. The 4-stage unrolling of machine *A* is shown in the expressions below. There are two different formulas for writing the output *q* in terms of outputs in previous cycles. These are obtained by repeatedly expanding each variable using its logic expression. The expressions after expansion are $F1_n = \overline{F1_{n-1}}$ and $F2_n = F1_{n-1} \oplus F2_{n-1}$ for all *n*. Therefore, Machine *A* is 3-inductive where *S=0* and $q_n = \overline{(q_{n-3} + q_{n-2} + q_{n-1})}$, and 4-inductive where *S=0* and $q_n = q_{n-4}$.

$$\begin{aligned}
q_n &= F1_n F2_n = \overline{F1_{n-1}}(F1_{n-1} \oplus F2_{n-1}) = \overline{F1_{n-1}}F2_{n-1} \\
&= F1_{n-2}(F1_{n-2} \oplus F2_{n-2}) = F1_{n-2}\overline{F2_{n-2}} = \overline{F1_{n-3}}(F1_{n-3} \overline{\oplus} F2_{n-3}) = \overline{F1_{n-3}}\ \overline{F2_{n-3}} \\
&= \overline{(q_{n-3} + q_{n-2} + q_{n-1})} \text{ where } q_{n-3} = F1_{n-3}F2_{n-3},\quad q_{n-2} = \overline{F1_{n-3}}F2_{n-3}, \\
&\qquad \text{and } q_{n-1} = F1_{n-3}\overline{F2_{n-3}} \\
&= F1_{n-4}(F1_{n-4} \overline{\oplus} F2_{n-4}) = F1_{n-4}F2_{n-4} = q_{n-4}
\end{aligned}$$

Machine *B* is a ring counter initialized at *F*1=1, *F*2=0, and *F*3=0. A 4-stage unrolling of machine *B* shows that it is 4-inductive with *S*=0 and $q_n = q_{n-4}$. Given the equal formulae $q_n = q_{n-4}$ for both machines, if the two machines produce similar outputs in the first 4 cycles, then by Theorem 2 they produce equal outputs thereafter, i.e. *A*=*B*.

cs2	cs1	ns2	ns1	q
0	0	0	1	0
0	1	1	0	0
1	0	1	1	0
1	1	0	0	1



cs4	cs3	cs2	cs1	ns4	ns3	ns2	N1	q
0	0	0	1	0	0	1	0	0
0	0	1	0	0	1	0	0	0
0	1	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	1

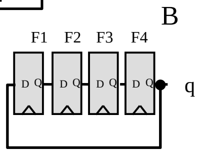


Figure 9: *4-inductive circuit with S=0*

Definition 6: *Let Machine A be m-inductive under set S. Suppose two sequential circuits A and B have the same PIs and POs. Use some 1-1 mapping between the FF's of A and B to form* B^1A^{k-1} *and* A^k *as shown in Figure 5. The equality* $[B^1A^{k-1} = A^k]^{sm}$ *for* $m \le k-1$ *means that all the corresponding POs of* B^1A^{k-1} *are equal to those of* A^k *and the nodes S in the* *last m copies of A^1 in B^1A^{k-1} are equal to corresponding nodes S in the last m copies of A^k .*

Lemma 3: *Let Machine A be m-inductive under set S. Suppose two sequential circuits A and B have the same PIs and POs. Use some 1-1 mapping between the FF's of A and B to form B^1A^{k-1} and A^k as shown in Figure 5. If $[B^1A^{k-1} = A^k]^{S^m}$ for $m \le k-1$, then $[B^pA^{k-1} = A^{p+k-1}]^{S^m}$ for any $p > 0$.*

Proof: We use induction on *p* to prove this. For the base case, this holds for $p=1$ by assumption. For the inductive case, assume it is true for some *p*, i.e. $[B^pA^{k-1} = A^{p+k-1}]^{S^m}$. Apply A^1 on the right to each side. Equality under S^m still holds since the corresponding nodes in *S* and outputs are equal in the last *m* copies of A^1 on both sides of the equation and *A* has the *m-inductive* property on S; Thus, $[B^pA^{k-1}A^1 = A^{p+k-1}A^1]^{S^m}$. Regroup A^1's, $[B^pA^1A^{k-1} = A^{p+k}]^{S^m}$. Replace $[A^1A^{k-1}]^{S^m}$ with $[B^1A^{k-1}]^{S^m}$ to get $[B^pB^1A^{k-1} = A^{p+k}]^{S^m}$. Regroup the B^1's, $[B^{p+1}A^{k-1} = A^{p+1+k-1}]^{S^m}$, and by induction, $[B^pA^{k-1} = A^{p+k-1}]^{S^m}$ for any $p > 0$. **QED**

Theorem 3: *Let Machine A have the m-inductive property under set S. Suppose two sequential circuits A and B have the same PIs and POs. Use some 1-1 mapping between the FF's of A and B to form B^1A^{k-1} and A^k . If $[B^1A^{k-1} = A^k]^{S^m}$, then A = B, when both are initialized with the same arbitrary initial state.*

Proof: Suppose the theorem is false meaning $[B^1A^{k-1} = A^k]^{S^m}$ but $A \neq B$. Then, there exists a state s_0 , an integer l, and a sequence of PIs, $\hat{I} = \{\hat{I}_1, \hat{I}_2, \ldots, \hat{I}_l, \ldots\}$ such that at clock cycle l, one of the POs of *A* differs from the corresponding PO of *B*, when both *A* and *B* are given the same initial state s_0 . From Lemma 3 $[B^pA^{k-1} = A^{p+k-1}]^{S^m}$ when $[B^1A^{k-1} = A^k]^{S^m}$ meaning outputs of *A* and *B* are equal for any p > 1. Choose p large enough such that $p > l$. This contradicts the existence of the sequence $\hat{I}$ that causes a difference in outputs. Thus, *A = B*. **QED**

Lemma 2 states that if *S* includes all *NS* outputs of a machine *A*, *A* is *1-inductive* under *S*. As a result, Theorem 1 in [14] is a special case of Theorem 3 in this paper where *m*=1 and *S* is the set of all *NS* nodes.

C. Optimization of m-Inductive Circuits

Consider a sequential circuit *A* which has the *1-inductive* property under some set *S*. This circuit is to be optimized using a *k*-step unrolling process where the first copy of A^1 in A^k is optimized using ODCs from the last *k*-1 copies of A^1. The optimized circuit is denoted as B^1. According to Theorem 3, if $[B^1A^{k-1} = A^k]^{S^1}$, then *A = B*. This means as long as A^k equals B^1A^{k-1} for all the outputs and all nodes *S* in the last copy of A^1, *A*= *B*. As a result, one only needs to preserve the behavior of the outputs and nodes in *S*. In particular, the *NS* functions that are not in *S* can be changed.

Now, let *A* be *m-inductive* under the set *S*. In this case, one needs to preserve nodes of *S* in the last *m* copies of A^1. In order to get a larger don't care set than *m*-inductive case, create the unrolling A^{k+m}. After optimization of the first copy of A^1 in A^{k+m} into B^1, the following must hold $[B^1A^{k+m-1} = A^{k+m}]^{S^m}$. Notice that the parameter *m* needed for *m*-inductiveness is different from *k*. The unrolling A^k can be increased or decreased independent of *m*.

Example: The state machine in Figure 8 shows a circuit *A* that is *1-inductive* under the empty set *S*. A *2-step* unrolling of this circuit, A^2, is also shown (the unrolling is for times *n-1* and *n*). The optimization needs to maintain only the outputs at time *n*-1 and *n* while synthesizing the first copy of A^1. It turns out that the signal r_n is redundant with respect to outputs q_{n-1} and q_n. Therefore, one instance of the AND gate can be removed. The new optimized circuit *B* is also shown in Figure 8. Notice that the next-state function at flip flop F_2 has changed and is independent of input *r*. This optimization cannot be done using previous sequential optimization techniques that only rely on an unrolling like the ones in [8].

Lemma 4: *Let Machine A be m-inductive under set S. The ODCs computed at any node in the first copy of A^1 in the unrolling A^{k+1} are larger than or equal to the ODCs computed under the unrolling A^k.*

Proof: Let N_i be any node in the first copy of A^1 under the unrolling A^k as shown in circuit C^1 in Figure 5, let g_i be the global function of that node in terms of inputs and the *CS* variables, and D_i be the ODC of that node in terms of inputs and *CS* variables. Any new function $\tilde{g}_i$ can replace g_i as long as $g_i - D_i \subseteq \tilde{g}_i \subseteq g_i + D_i$. This change in g_i will not be observable at the POs and the subset *S* in the last *m* copies of A^1 in A^k. Since this change is not observable at A^k, it will not be observable A^{k+1} since *A* is *m*-inductive. As a result the ODC of N_i computed from A^k must be a subset of the ODC of N_i computed from A^{k+1}. **QED**

Lemma 5: *Let Machine A have the m-inductive property under the NULL set for S. The ODCs computed at any node in the first copy of A^1 in the unrolling A^{m+1} are equal to those computed in the unrolling A^m.*

Proof: Let N_i be any node in the first copy of A^1 in the unrolling A^{m+1}, and g_i and D_i be the global function and ODC respectively of that node in terms of inputs and *CS* variables. Let $\tilde{g}_i$ be any function such that $g_i - D_i \subseteq \tilde{g}_i \subseteq g_i + D_i$. Thus, changing g_i to $\tilde{g}_i$ is not observable at any of the POs of A^{m+1}. Since *A* is *m*-inductive under *S*=NULL, the change $\tilde{g}_i$ being not observable at the outputs of A^m, already implies it is not observable at the outputs of A^{m+1}. Hence the extra copy of A^1 could not have created additional don't cares. Thus the SODCs of A^m and A^{m+1} are equal when *S* is NULL. **QED**

Lemma 5 states that if *S* is NULL, there is no need to unroll a circuit *A* that is *m-inductive* under *S* more than *m* times. Additional unrolling does not result in a larger ODC set, e.g. in Figure 8, unrolling beyond A^2 does not give more ODCs.

D. Pipelines

The state machine in Figure 10 shows a pipeline circuit *A* with loops where the next-state functions of FF_1 are dependent on the state variables of FF_m. Each stage of the pipeline has its own inputs and outputs. Figure 10 also shows the (m+1)-*step* unrolling of A from time n-m to n. It can be observed from the unrolling that the next-state functions of any column of flip-flops FF_i and all outputs at time n can be written in terms of inputs from n-m+1 to n and state variables of FF_i at time n-m for any $1 \le i \le m$. Therefore this circuit is *m-inductive* when set S includes all flip flops of any stage FF_i. For the case, where S includes all the flip flops of stage m (FF_m), an optimization of the first copy of A^1 in A^{m+1} needs to maintain the global next-state functions of FF_m in the last m copies of A^1 in A^{m+1} (outputs of grey boxes in Figure 10) in addition to global functions of outputs in the unrolling A^{m+1}.

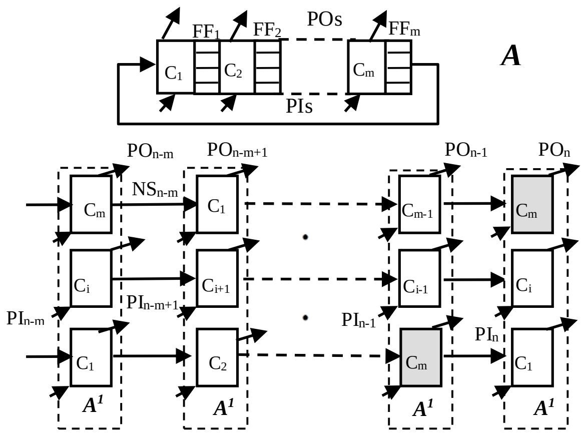


Figure 10: *m-stage pipeline with feedback loop*

An *m-stage* pipeline circuit without loops is formed when all the loops and all signals coming from or going to the rest of the circuit are cut. The cut points form the inputs and outputs of the pipeline. In this case, the Boolean functions of the outputs of the pipeline at time n can be described in terms of PIs from time n-m+1 to n. This description is independent of state variables. Therefore using Theorem 1, an m-stage pipeline without loops is *m-inductive* with S equal null.

IV. FINDING M-INDUCTIVE SETS

We describe a method for finding a minimal subset S of next-state functions of a sequential circuit to make it *m-inductive* under S. Each flip-flop is interpreted as a pair of nodes, one being the flop input, the other the flop output. The algorithm finds a set S which is a subset of the flop inputs.

Consider Figure 7 showing the unrolling A_x and A_y of a sequential circuit A for m+1 timeframes. This unrolling illustrates the construction of an instance of Boolean satisfiability (SAT), which is used to compute S. To construct the SAT instance, we use $2 \times (m+1)$ copies of the Conjunctive Normal Form (CNF) for the combinational logic of the sequential circuit, representing m+1 timeframes of the unrolling of the two identical sequential circuits.

We also use n auxiliary variables to represent the next-state functions of each flip-flop. The auxiliary variables have the meaning that the variable is 1 if and only if the corresponding next-state function belongs to S. In addition, the following "glue" constraints are added to the SAT instance.

1. Clauses forcing equality of flop outputs and flop inputs for both copies of A (i.e. A_x and A_y). These force equality of flop outputs at time $i+1$ with flop inputs at time i for any $1 \le i \le m$.
2. Clauses forcing equality of corresponding primary outputs in A_x and A_y for any time i, $1 \le i \le m$.
3. Clauses forcing equality of corresponding nodes in S for A_x and A_y for any time i, $1 \le i \le m$.
4. Clauses forcing inequality of at least one pair of primary outputs or nodes in S for A_x and A_y in the m+1 timeframe.

Auxiliary variables are used only in the constraints 3 and 4. Specifically in 3, each equality of the corresponding pair of S-nodes (n_x, n_y) from A_x and A_y at time i is written as two implications $n_x \Rightarrow n_y$ and $n_y \Rightarrow n_x$ with the additional enabling auxiliary variable h, resulting in two clauses: $(\bar{n}_x + n_y + \bar{h})(n_x + \bar{n}_y + \bar{h})$. Thus, the implications are forced to hold only if $h = 1$. However, if $h = 0$, these clauses are immediately satisfied and the equality of the pair of S-nodes is not required. A similar construction is used in generating constraints 4.

The SAT instance constructed above can be used to evaluate if a given set S of next-state functions is *m-inductive*. For this, we use a set of assumptions, which force the given set of next-state functions to be included in S. The assumptions are derived as follows: each auxiliary variable is assigned value 1 (value 0) iff the corresponding next-state function is included in (excluded from) S. The resulting SAT instance is UNSAT iff the corresponding set S is m-inductive.

Note that the same SAT instance can be used to check *m-inductiveness* of different sets of flops in S. For this, only the set of assumptions needs to be changed. This observation is used to efficiently implement the algorithm shown in Figure 11 for finding a minimal *m-inductive* set of next-state functions.

```
Input: Sequential design, value of m
Outputs: A minimal set of next-state functions S
Initialize S to contain all next-state functions.
Create SAT instance for given S and m, as described above.
Assert that this S is m-inductive (should be always true).
for each node s belonging to S {
   S = S – s;
   If ( the resulting S is not m-inductive )
      S = S + s;
}
return S;
```

Figure 11: *Pseudo-code to compute minimal m-inductive set.*

V. EXPERIMENTAL RESULTS

The proposed algorithm for finding a minimal set S of next-state functions for m-inductiveness was implemented in ABC and run on multiple ISCAS and MCNC benchmark circuits. Table 1 shows experimental results on 10 designs. The first column shows the names of the circuits. The second column shows the number of flip flops FF, primary inputs I, and primary outputs O for each design. Columns 3 to 7 show s the cardinality of set S computed by the algorithm for $1 \le m \le 5$, V the number of variables used to set up the SAT problems, and T the runtimes in seconds to compute S.

The computation of a minimal set S starts with m=1. Once found, it is used as a starting set for m=2. This iteration continues until m=5. The last row in the table has the total number of flip flops in all designs and the total number of elements in set S for each value of m. The number of elements in S is 16% smaller than the total number of flops when m = 1. The largest percentage drop in next-state functions is when m is increased from 1 to 2. In this case, the number of elements in *S* is 42% smaller than the total number of flip flops. When *m* goes from 4 to 5, the percentage reduction in size of *S* is only 2%. Using values greater than 5 for *m* does not materially reduce the size of *S*. As shown in the case of data paths, there is no unique solution for *S* and some minimal sets can be smaller than the others. The table shows the computation for one minimal set per design; it does not enumerate all possible solutions.

NAME	FF/I/O	M=1	M=2	M=3	M=4	M=5
s13207	FF=669 I=31 O=121	s=392 V=11602 T=3s	s=256 V=15944 T=2s	s=218 V=20286 T=2s	s=197 V=24628 T=2s	s=188 V=28970 T=2s
s9234	FF=211 I=36 O=39	s=176 V=4622 T=1s	s=148 V=6472 T=1s	s=140 V=8322 T=1s	s=130 V=10172 T=2s	s=129 V=12022 T=1s
s35932	FF=1728 I=35 O=320	s=1728 V=34376 T=33	s=848 V=47788 T=276s	s=699 V=61200 T=191	s=499 V=74612 T=367s	s=470 V=88024 T=847s
s38417	FF=1636 I=28 O=106	s=1432 V=31040 T=92s	s=1079 V=43182 T=115s	s=1023 V=55324 T=177s	s=978 V=67466 T=256s	s=969 V=79608 T=334s
key	FF=228 I=258 O=193	s=76 V=7562 T=2s	s=39 V=10694 T=2s	s=39 V=13826 2s	s=39 V=16958 T=3s	s=39 V=20090 T=3s
Min-max32	FF=96 I=35 O=32	s=32 V=2872 T=1s	s=32 V=4084 T=2s	s=32 V=5296 T=2s	s=32 V=6508 T=5s	s=32 V=7702 T=4s
Cpb.-32.4	FF=32 I=33 O=33	s=0 V=1066 T=0.1s	s=0	s=0	s=0	s=0
s641	FF=19 I=35 O=23	s=15 V=678 T=0.01s	s=8 V=956 T=0.01s	s=8 V=1234 T=0.01s	s=8 V=1512 T=0.01s	s=8 V=1790 T=0.01s
s1423	FF=74 I=17 O=5	s=73 V=1540 T=0.1s	s=71 V=2157 T=0.1s	s=69 V=2774 T=0.1s	s=68 V=3391 T=0.1s	s=68 V=4008 T=0.2s
s38584	FF=1452 I=12 O=278	s=1239 V=31136 T=38s	s=1096 V=43522 T=59s	s=1035 V=55908 T=190s	s=981 V=68294 T=298s	s=947 V=80680 T=552s
Total FF	6145	s=5163 84%	s=3577 58%	s=3263 53%	s=2932 48%	s=2850 46%

Table 1: *Finding a minimal S for different values of m*
All the runtimes are on a single thread of a 2.2 GHZ Intel i7 machine.

In another experiment, 1-inductive properties of a design are used to optimize its combinational logic. First a minimal set *S* is computed for a design *A* so that it is 1-inductive under *S*. Each design *A* is then unrolled twice to form A^2. A logic optimization algorithm is applied to optimize the first copy of A^1 in A^2 to form B^1A^1. The optimization algorithm maintains the logic functions of all outputs of A^2 and those next-state functions in *S* of the last copy of A^1 in A^2. The next-state functions not in *S* are not unconstrained. (One can think of these as floating outputs or consider their external don't care to be the full Boolean space.) The optimization algorithm applies existing algorithms in ABC to use ODCs from the second copy of A^1 in A^2 to optimize the first copy to create a new design B^1. B^1 is the combinational logic of a new sequential machine *B* which is sequentially equivalent to *A* since $[B^1A^1 = A^2]^S$.

The algorithm is implemented in ABC and is applied to a large set of ISCAS and industrial designs. To judge the additional power of this optimization, we compared a) the literal count (in factored form) of each design when optimized combinationally, to b) the case when optimized using 1-inductiveness. On average, 25% of designs showed good improvement when 1-inductiveness was used.

Table 2 shows some of the designs where good improvement was obtained. The number of flops in *S* and the total number of flops for each design are shown in the second column of Table 2 (labeled *S/FF)*. Column 3 (labeled *INIT*) shows the initial count of each circuit measured in factored form literal count. Column 4 (labeled *COMB*) shows the literal count when only combinational optimization is used and Column 5 shows literal count and runtimes for optimization using 1-inductiveness. The percentage improvement in literal count (1-IND vs COMB) is shown in Column 6. Some industrial designs show significant reduction in literal count when inductiveness was used, e.g. Ind1 shows 7% reduction in literal count. Notice that this extra sequential optimization is obtained independent of the initial states of the design. Also note that the circuits that show the most improvement in literal count also show big reductions in the sizes of *S* relative to the total FFs in the design.

When the first copy of A^1 in A^2 is optimized, some flexibility in optimization may come from ODCs in the second copy of A^1 and some from the fact that *S* is a subset of the FFs. To determine the contribution from ODCs, we set *S* to be all the flops (all NS functions are maintained in A^2) and re-ran the optimization on all designs. Since the resulting final literal count of each design was very close to the combinational results, we conclude that most of the improvement in literal count shown in Table 2 is due to the reduction in the size of *S*.

NAME	S/FF	INIT	COMB	1-IND/T	%LIT-R
s13207	392/669	4007	3789	3338 / 6s	11.9
s9234	176/211	2582	2402	2140 / 1s	12.4
s38417	1432/1636	13418	13234	13166/212s	0.5
Ind1	598/1140	8863	7083	6589 / 181s	7
Ind2	257/669	19517	19384	19017 / 65s	1.9
Ind3	150/402	15040	14880	14513 / 36s	2.5
Ind4	514/594	5455	5322	5188/7s	2.5
Ind5	514/594	5425	5292	5158 / 7s	2.5
Ind6	349/426	3474	3458	3376 / 3s	2.4
Ind7	125/129	1312	1312	1292 / 0s	1.5

Table 2: *Optimizing 1-inductive circuits*

Another set of experiments was run to find the sets *S* when 2-inductiveness was used to optimize the first copy of A^1 in A^3. In this case, the next-state functions of the second and third copies of A^1 in A^3 that are in *S* need to be maintained. The comparison of 1-inductive and 2-inductive optimization is shown in Table 3. Although, one can build special circuits where only 2-inductive optimization reduces literal count, we did not see much improvement compared to 1-inductive optimization in the set of designs we tested.

NAME	FF	S1	1-IND	S2	2-IND
s13207	669	392	3338	256	3347
s9234	211	176	2140	148	2134
s38417	1636	1432	13166	1096	12977
Ind1	1140	598	6589	596	6596
Ind2	669	257	19017	5	19019
Ind3	402	150	14513	9	14513
Ind4	594	514	5188	389	5173
Ind5	594	514	5158	398	5133
Ind6	426	349	3376	241	3376
Ind7	129	125	1292	108	1292

Table 3: *Comparing 1-inductiveness and 2-inductiveness*

VI. Conclusions

A new concept, *m*-inductiveness over a minimum set of signals *S*, is introduced. It can be used for synthesis and verification of sequential machines. It is shown that *S* is related to the structure of the circuit. Our SAT-based algorithm finds a minimal set of signals *S* to be used for *m*-inductiveness. Experimental results show that for *m*=1, *S* is about 16% smaller than the total FFs. It is also shown, that an optimization algorithm that uses the flexibility of *m*-inductiveness can significantly reduce the area of a design.

References

[1] J. Baumgartner, H. Mony, V. Paruthi, R. Kanzelman and G. Janssen, "Scalable sequential equivalence checking across arbitrary design transformations." *Proc. ICCD'06*.

[2] Berkeley Logic Synthesis and Verification Group. *ABC: A System for Sequential Synthesis and Verification*.

[3] S. Chatterjee, A. Mishchenko, R. Brayton, and A. Kuehlmann. "On resolution proofs for combinational equivalence", *Proc. DAC'07*.

[4] A. P. Hurst, "Automatic synthesis of clock gating logic with controlled netlist perturbation", *Proc. DAC'08*, pp. 654-657.

[5] M. A. Iyer, D. E. Long, and M. Abramovici, "Identifying sequential redundancies without search," *Proc. DAC'96*, pp. 457-462.

[6] M. A. Iyer, D. E. Long, and M. Abramovici, ‘‘Surprises in sequential redundancy identification,’’ *Proc. EDTC'96*.

[7] J.-H. R. Jiang and W.-L. Hung. "Inductive equivalence checking under retiming and resynthesis". Proc. ICCAD'07, pp. 326-333.

[8] A. Mehrotra, S. Qadeer, V. Singhal, R. K. Brayton, A. Aziz, and A. L. Sangiovanni-Vincentelli. "Sequential optimization without state space exploration". *Proc. ICCAD'97*, pp. 208-215.

[9] A. Mishchenko and R. K. Brayton, "Recording synthesis history for sequential verification", *FMCAD'08*, pp. 27-34.

[10] H. Mony, J. Baumgartner, V. Paruthi, and R. Kanzelman, "Exploiting suspected redundancy without proving it". *Proc. DAC'05*.

[11] A. Mishchenko, S. Chatterjee, R. Brayton, and N. Een, "Improvements to combinational equivalence checking", *Proc. ICCAD '06*, pp. 836-843.

[12] A. Mishchenko, M. L. Case, R. K. Brayton, and S. Jang, "Scalable and scalably-verifiable sequential synthesis", *Proc. ICCAD'08*, pp. 234-241.

[13] A. Saldanha, A. Wang, R. K. Brayton, and A. Sangiovanni-Vincentelli, "Multi-level logic optimization using don't cares and filters", *Proc. DAC'89*.

[14] H. Savoj, A. Mishchenko, and R. Brayton, "Combinational techniques for sequential equivalence checking". *Transactions on Computer Aided Design of Integrated Circuits Systems (Volume 33 Issue 2)*, Feb. 2014, pp 305-317.

[15] H. Savoj, "Don't Cares in Multi-Level Network Optimization". PhD Thesis, University of California Berkeley.

[16] V. Singhal and C. Pixley. "The verification problem for safe replaceability

[17] ," *Proc. CAV'94*, LNCS, Vol.818, pp. 311-3