Scalable Sequential Verification

Robert Brayton Alan Mishchenko

EECS Department, University of California, Berkeley, CA 94720
{brayton, alanmi}@eecs.berkeley.edu

Abstract: In general, sequential verification is PSPACE complete, but for application to present-day industrial designs, it needs to be made scalable, which means essentially linear in circuit size. This paper focuses on the problem where the circuit in question has been transformed using a form of scalable sequential synthesis. During this synthesis, a history And-Inverter-Graph (HAIG) is constructed, which efficiently records all logic nodes ever created in the synthesis process. A HAIG can be constructed in a fast, memory efficient, and scalable way. It is an FSM, which contains the initial and final FSMs to be compared as well as many redundant, sequentially equivalent nodes. The sets of equivalent nodes form "bridges" connecting the initial and final machines and can be used to construct an inductive invariant. It is shown that this invariant is sufficient to prove delayed sequential equivalence between the two FSMs. The complexity of validating this is roughly the cost of one combinational SAT call of the size of the two machines; however, the many structural similarities, which are pre-identified in the HAIG structure, make the proof of invariance particularly easy and scalable.

1 Introduction

Sequential synthesis can result in considerable reductions in delay (e.g. see [15]); however, most present-day industrial CAD tools do not use sequential synthesis for reasons of non-scalability of both synthesis and verification. The present paper is based on the premise that seqential synthesis and verification must go hand-in-hand to make sequential synthesis acceptable.

For scalable sequential synthesis, "sequential rewriting" [17] was proposed, which can be decomposed into small steps of retiming and resynthesis [14]. This extends rewriting to the sequential case in a transparent way, which makes it just as efficient as combinational rewriting [16]. We review sequential rewriting in Section 3.

Much work has been done on sequential equivalence checking (SEC) of two FSMs. Generally, the problem is PSPACE complete; however, if synthesis is restricted by one set of combinational transformations followed by one retiming or vice versa, the problem is provably simpler. On the other hand, the addition of only one additional set of combinational transformations makes the problem again PSPACE complete [12]. Also, like combinational equivalence checking (CEC), the problem becomes simpler in practice if there are structural similarities between the two circuits to be compared.

There are also different forms of equivalence for FSMs. The two relevant ones for this paper are the traditional equivalence[1] between two given initial states, and *delayed equivalence* [20], which states that, in a certain finite number of clock cycles after power up, for each state in one machine, there exists an equivalent state in the other machine.

The presented work comes closest to the following two approaches in the literature. Van Eijk [9] derived an inductive invariant, constructed by a fixed point process, consisting of a set of equivalences between signals in the two circuits. This invariant characterizes a superset of the reachable states of the product machine. Bjesse [4] and Case [7] extended this to an invariant composed of implications. These methods are dependent on the particular implementation structures of the two machines because equivalences or implications can be detected only between existing signals. To overcome this limitation, Van Eijk proposed creating additional

[1] Equivalence of two states is the usual notion that for any input sequence the output sequences produced by starting in the two states are identical.

signals, without any fanout, which might be useful in establishing more equivalences. His proposal involved adding nodes which could be obtained by retiming. These methods approximate the reachable state space, thereby simplifying SEC, but do not guarantee that the invariant derived is sufficient to prove sequential equivalance.

Mneimneh et. al. [18] looked at the problem of one retiming and one set of combinational logic transformations (in either order) and proposed a *retiming invariant* composed of a conjunction of functional relations among latch values derived from the atomic retiming moves. They described how to construct this invariant when only retiming was involved.

We address a more general problem when one machine is derived from the other by an arbitrary sequence of interleaved combinational rewritings, duplications, and retimings[2]. In contrast to the method of van Eijk, our approach can be characterized as follows:

1. *All* nodes created during synthesis,are recorded instead of adding a set of ad hoc signals.
2. The invariant is written down *directly* instead of being derived by a fixed-point iteration..
3. The invariant is sufficient to prove delayed sequential equivalence of the two machines.

This invariant is the conjunction of all *rewriting invariants* (developed in Section 4) that are recorded after performing each sequential rewriting step.[3] Finally, we show how the invariant can be used to derive an initial state of the final machine equivalent to the given initial state of the original machine.

Organization. Section 2 surveys combinational and sequential AIGs. Section 3 gives an overview of sequential rewriting. Section 4 gives some theoretical background and results for proving sequential equivalence. Section 5 describes the history AIG, the concept of sequential choice nodes, and gives the details of the inductive invariant. Section 6 provides the theory behind the inductive invariant, and gives two methods for proving sequential equivalence using the HAIG. Section 7 details how the HAIG and the inductive invariant can be used to derive a new equivalent initial state. Section 8 gives some experimental results and Section 9 concludes.

2 Sequential and Combinational AIGs

A (combinational) *And-Invertor Graph* (AIG) is a Boolean network composed of two-input ANDs and inverters. *Structural hashing* of AIGs ensures that, for each pair of nodes, all constants are propagated and there is at most one AND node having them as fanins (up to permutation). It is performed by one hash-table lookup when AND nodes are created and added to an AIG manager. An AIG is often *balanced,* to reduce the number of AIG levels*,* by applying the associative transform, $a(bc) = (ab)c$. Both structural hashing and balancing are performed in one topological sweep from the PIs and have linear complexity in the number of AIG nodes.

The *size* (*area*) of an AIG is the number of its nodes; the *depth* (*delay*) is the number of nodes on the longest path from the PIs to the POs. The goal of optimization by local transformations of an AIG is to reduce both area and delay.

Definition. A *cut* C of a node n is a set of nodes of the network, called *leaves* of the cut, such that each path from a PI to n passes through at least one leaf. Node n is called the *root* of cut C. The cut *size* is the number of its leaves. A *trivial cut* of a node is the cut composed of the node itself. A cut is *K-feasible* if the number of nodes in the cut does not exceed K. A cut is *dominated* if there is another cut of the same node, which is contained, set-theoretically, in the given cut. The *volume* of a cut is the total number of nodes encountered on all paths between node n and the cut leaves.

Definition. A *local* function of an AIG node n, denoted $f_n(x)$, is a Boolean function of the logic cone rooted in n and expressed in terms of the leaves, x, of a cut of n. The *global function* of an AIG node is its function in terms of the PIs of the network.

[2] Sequential rewriting can be decomposed into these operations.

[3] When only retiming is involved in the rewriting step, the rewriting invariant coincides with this retiming invariant of [18].

Definition. *Exhaustive simulation* of a cut with k leaves is (bitwise) simulation of all 2^k different input patterns. This produces the truth table of a local function of the cut root in terms of the cut leaves. Exhaustive simulation can be used to efficiently compare logic functions when k does not exceed 16.

Sequential AIGs add sequential elements to the structure of combinational AIGs. The sequential elements are technology-independent D-latches with one input and one output and controlled by the same clock, which is omitted in the AIG representation.

We represent latches in the AIG explicitly as one-input "boxes".[4] Structural hashing of sequential AIGs [1] is similar to that for combinational AIGs. A node is a PI, a PO, an AND-gate, *or* a latch. Structural hashing maintains the invariant that no two nodes have the same fanins. In addition, two invariants are enforced to make sequential AIGs "more" canonical. The additional degree of canonicity achieved by this is relatively small, but scalable and efficient, and results in more compact AIGs.

1. A latch cannot have a complemented fanin. If one is about to be created during the AIG construction, the complemented attribute of the fanin edge is propagated to the fanout and the initial value of the latch is complemented.
2. An AND-node cannot have two latch fanins. If one is about to be created during the AIG construction, the fanin latch pairs are retimed forward over the AND-node.

Thus, structural hashing may propagate changes to the fanouts and lead to rehashing large portions of the AIG. For example, retiming a latch forward over a node may result in a sequence of other retimings in order to maintain Invariant 2.

Definition. A *cut* in a sequential AIG for a node n, is a set of AND nodes, latches, or PIs in its transitive fanin, TFI(n), which separates the cone between the cut and n from the rest of its transitive fanin network, i.e. the cone between the cut and n has no other inputs.

Two properties of this cut are: (1) the cone cannot contain a loop, and (2) given values for the cut variables and the latches within the cone, the value of node n is uniquely determined.

Definition. The *leaves* of a cut are an indexed set of the cut nodes, where each node may be multiply indexed by the number of latches appearing on any path from the root to the node.

For example, if node b is in a cut and there is a path from b to the root n with no latches, and another path from b with one latch, then the cut leaves include independent variables b^0 and b^1. It is always possible to express a local function at node n as a *combinational* function of the leaf variables of a cut for node n. For convenience, we will refer to a set of cut leaves in a sequential AIG as a *sequential cut*.

3 Sequential Rewriting

3.1 Rewriting of Sequential AIGs

DAG-aware rewriting [16] for combinational AIGs comprises a set of fast greedy algorithms to minimize AIG size. One such algorithm considers all 4-input cuts rooted at each node. For each cut, it tries to replace the current logic structure of the cut cone by one of a set of pre-computed logic structures implementing the same logic function at the root. If there is an improvement in the number of AIG nodes (considering sharing with existing nodes) and (optionally) no increase in the number of AIG levels, the current logic structure is modified accordingly, and the computation moves to the next node in a topological order. A detailed discussion of the algorithm is found in [4][16].

This algorithm was modified for sequential circuits to use sequential cuts which cross latch boundaries. To evaluate such a cut, the latches are moved (conceptually) backwards to the leaves of the cut, while performing logic duplication if necessary, as shown in the example below. The combinational logic function of the root in terms of the resulting leaf variables is

[4] Previous work on sequential AIGs [1][15] represented latches and their initial values as attributes on AIG edges, similar to the work of Leiserson and Saxe [13].

then rewritten in a new form (exactly as done in the combinational case). In assessing the gain, the new logic used to replace the old logic is structurally hashed to move latches as far forward as possible and to share as much logic as possible.

Sequential rewriting for a given node involves the following conceptual atomic steps (illustrated in Figure 3.1):

1. compute a sequential cut for a node,
2. duplicate nodes on the reconvergent paths with different numbers of latches and assign these nodes different indices,
3. retime all latches backward to the cut leaves,
4. do combinational rewriting on the resulting combinational function in terms of the cut leaf variables,
5. structurally hash the result.

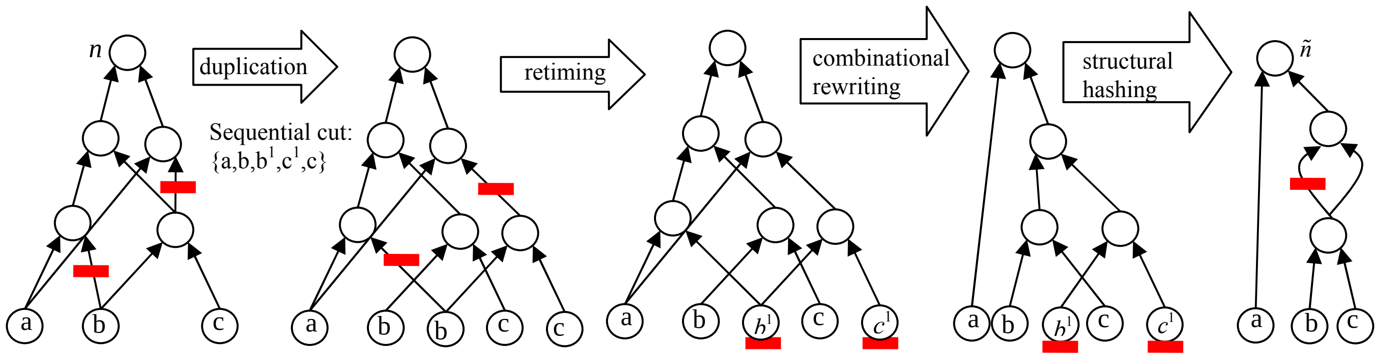

Figure 3.1. The conceptual steps of sequential rewriting illustrated.

If the same variable appears in more than one time frame, duplication is necessary to be able to retime all latches to the inputs of the cut. After pure combinational rewriting of the cone logic, all latches are retimed as far forward as possible in order to "canonicize" the sequential AIG to share more logic.[5]

4 Theoretical Framework

4.1 Sequential Equivalence

Definition: Two nodes x and y are SE ($x \overset{SE}{\approx} y$) if they have identical combinational logic functions of a common sequential cut.

For example, the two root nodes, n and $\tilde{n}$, in Figure 3.1 are SE because they share the same function of a common cut, namely $f_n = abb^1cc^1 = f_{\tilde{n}}$.

Definition: *Sequential depth* is the largest index on the sequential cut variables.

For example, the cut $\{a,b,b^1,c^1,c^2\}$ has sequential depth 2.

Definition: Two states are *equivalent*, $s_X \equiv s_Y$, if for any input sequence, the resulting output sequences, starting from the two states, are identical.

[5] The last two steps effectively constitute sequential structural hashing as discussed in Section 2

Definition: Machine X *implies* machine Y ($X \Rightarrow Y$) if $\forall s_X \in S_X, \exists s_Y \in S_Y, s_X \equiv s_Y$, where S_X denotes the states of X etc.

Definition: C^N is the FSM that machine C becomes after it has been clocked for N cycles, i.e. C^N is the FSM with the set of states that can be reached in C after N or more clock cycles.

Definition: C is *delayed sequentially equivalent* to D if there exists N such that $C^N \Rightarrow D$.

Definition: Two machines are *synchronized* (*or aligned*) if they are in equivalent states.

Theorem [20]: If circuit C is derived from D by retiming, then there exists an integer N such that $C^N \Rightarrow D$.

Theorem [12]: If circuit C is derived from D by a sequence of retiming and resynthesis steps, then there exists an integer N such that $C^N \Rightarrow D$.

Since any sequential rewriting step consists of just retiming and resynthesis steps, then after any rewriting step transforming D_i to D_{i+1} , there exists N such that $D_{i+1}^N \Rightarrow D_i$. Similarly, a sequence of rewritings transforming D into C implies there exists N such that $C^N \Rightarrow D$.

4.2 Immediate Equivalence and an Invariant

After a rewriting step, the two associated root nodes, n and $\tilde{n}$, are SE by definition. This does not mean that their values are always equal. In general, the root node values are combinational functions of some of the values of the set of latches r, which are between the roots and the cut, plus some of the cut variables, x. In particular, they are functions of those latches and cut variables that can be reached from the roots without passing through a latch. We can express the values of the two root nodes combinationally as $n = g(r,x)$ and $\tilde{n} = \tilde{g}(r,x)$ where some of the r and x may not appear. Generally, $g(r,x) \neq \tilde{g}(r,x)$ for arbitrary values of (r,x), but we will derive a relation (called a *rewriting invariant*) among the latch values, r, which implies the equality at the root nodes. In Section 6, this relation will be the basis for an inductive invariant that is sufficient to prove delayed sequential equivalence and in Section 7, we use this invariant to filter rewriting steps so that an equivalent initial state for the synthesized machine can be computed from a given initial state of the original machine.

The rewriting invariant is derived as follows.

1. For a retiming step with sequential depth $k \geq 1$, conceptually retime all latches to its sequential cut.
2. Assign a symbolic parameter to each such latch to represent its value and let π_w denote these parameters.
3. Retime all such latches forward and derive the resulting latch values symbolically in terms of π_w , e.g. if r_i^w is a forward retimed latch of this rewriting step, its value is given by $r_i^w = f_i(\pi_w)$.
4. The rewriting invariant for step w is $I_w(r^w) = \exists_{\pi_w} \prod_i [r_i^w = f_i^w(\pi_w)]$.

Theorem 4.1: Let n_w and $\tilde{n}_w$ denote the two roots nodes of rewriting step w. Then, $I_w \Rightarrow (n_w = \tilde{n}_w)$.

Proof: By definition, $I_w = \exists_{\pi_w} \prod_i (r_i^w = f_i(\pi_w))$. Let π_w be such a set of values. During the rewriting of n_w into $\tilde{n}_w$, the functions at n_w and $\tilde{n}_w$ expressed in terms of the sequential leaf variables, x_w , are equal, i.e. $n_w = F(x_w) = \tilde{n}_w = \tilde{F}(x_w)$, since this is the result of combinational rewriting. Some of the parameters π_w may be directly related to x_w . This functionality is represented by $\pi_w^j = h_j(x_w)$. Also, the functions at n_w and $\tilde{n}_w$ can be expressed locally as functions of their *immediate* latch variables and possibly some of the leaf variables,

$$n_w = g(r^w, x_w)$$
$$\tilde{n}_w = \tilde{g}(r^w, x_w).$$

Using $r_i^w = f_i(\pi_w)$ and $\pi_w^j = h_j(x_w)$ to replace all variables except x^w, we obtain

$$n_w = g(r^w, x_w) = F(x_w)$$
$$\tilde{n}_w = \tilde{g}(r^w, x_w) = \tilde{F}(x_w)$$

which implies $n_w = \tilde{n}_w$, since $F = \tilde{F}$ as a result of rewriting. **QED**

Theorem 4.2: I_w is an inductive invariant of the sub-circuit C^w, composed of the TFI cones between the roots, n_w and $\tilde{n}_w$, and the cut, where the cut variables are the PIs and the POs are the root nodes..

Proof: We will use r^w to denote the variables that are the outputs of the latches and s^w to denote the variables that are the inputs to the latches. Suppose that the values currently in the latches r^w, satisfy $I(r^w) = 1$. Then there exists values π_w such that for any latch $r_i^w \in r^w$, $r_i^w = f_i^w(\pi_w)$. We need to show that there exists another set of values $\tilde{\pi}_w$ such that $s_i^w = f_i^w(\tilde{\pi}_w)$. The function $f_i^w(\pi_w)$ was obtained by propagating latch values forward, following the retiming moves. It can be obtained also by backward substitution, by the following process, where we use the notation $f(x)^+$ to denote that the index of all arguments is increased by 1.

1. Write down the combinational function for s_i^w in terms of its immediate latches, i.e. only those latches of r^w in a combinational path to s_i^w and some of the PIs of C^w (we called this function $g_i^w(r^w, x)$).
2. Compute $g_i^w(r^w, x)^+$ and replace each $x_i^j \in x$ by its corresponding π variable to obtain $\tilde{g}_i^w(r^w, \pi)$.
3. For any latch variable r_j^w, in the argument of $\tilde{g}_i^w(r^w, \pi)$, substitute $g_j^w(r^w, x)^+$ where each $x_i^j \in x$ has been replaced by its corresponding π variable.
4. Continue this recursively until no latch variables remain.

However, in Step 3, since the invariant holds, then $r_j^w = f_j^w(\pi_w)$. Thus by substituting $f_j^w(\pi_w)$ for r_j^w in $\tilde{g}_i^w(r^w, \pi)$, we get $(s_i^w)^+ = f_i^w(\tilde{\pi}_w)$. The sets of variables $\tilde{\pi}_w$ and π_w differ in that some variables are shifted by one unit of time. Nevertheless, there exists $\tilde{\pi}_w$ such that $(s_i^w)^+ = f_i^w(\tilde{\pi}_w)$ for all $r_i^w \in r^w$ proving that *I* is an inductive invariant. **QED**

Theorem 4.3: Let *d* be the sequential depth of rewriting step *w*. Then, starting from any arbitrary state *q* of C^w, any resulting state of C^w, after *d* clock cycles, will satisfy I_w.

Proof: C^w is a pipelined circuit (i.e. loop-free), so that any initial state will be flushed out in *d* clock cycles, at which point each latch will have a value computed by the combinational function given by the fanin cone from the latch input to the sequential cut. This is the same function required to satisfy I_w. **QED**

5 Choice Nodes and the History AIG

5.1 Sequential Choices

Whenever a rewriting step is executed on the current AIG (called the working AIG), the resulting sub-graph replaces its counterpart in the working AIG and is also copied and structurally hashed into a history AIG (HAIG), where the original node and its rewritten form are grouped together using a *choice node*. Previously, choice nodes referred only to nodes that were combinationally equivalent [8]; however for sequential rewriting, we can extend the notion of choice nodes to the $\overset{SE}{\approx}$ relation. The HAIG, which stores a copy of all nodes seen during synthesis, is the basis for scalable sequential verification (Section 7).

A HAIG is a sequential AIG with choice nodes. It should be noted that because of the choice nodes, a node in a HAIG can have several transitive fanin cones, and a cut is defined as a separator of any of these cones. A cut still has the property that a local function for a node *n* can be expressed as a combinational function of the leaf variables of the cut. At a choice node, its set of cuts is simply the *union* of the sets of cuts of the nodes of the choice class [8].

The process of recording rewriting steps in the HAIG is illustrated in Figure 5.1.

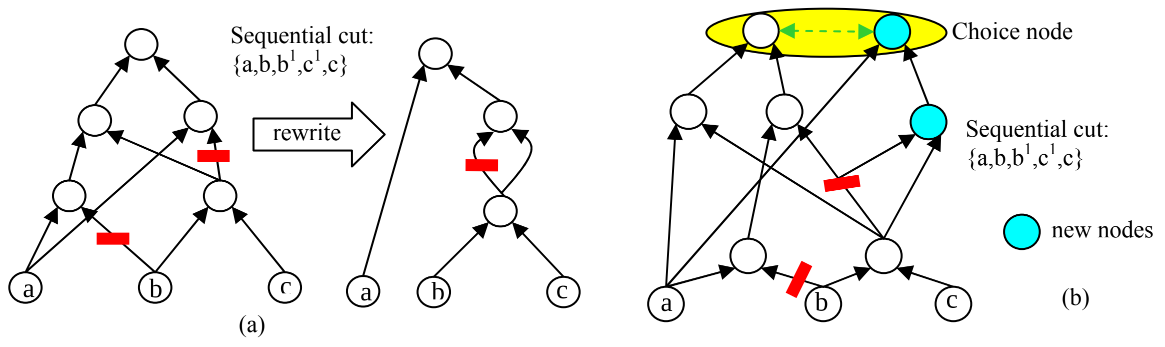

Figure 5.1. The HAIG accumulates choices. (a) Rewriting step. (b) HAIG after rewriting. Blank nodes denote pre-existing nodes and shaded ones denote new nodes.

After the structure created by rewriting is copied into the HAIG, structural hashing is used to propagate latches forward and to remove duplicate logic. Thus, in Figure 5.1b, the node with fanins *b* and *c* has only one copy when the rewriting result is added to the HAIG.

Theorem 5.1: $\overset{SE}{\approx}$ is an equivalence relation.

Proof. Clearly $\overset{SE}{\approx}$ is reflexive and symmetric. We prove transitivity, i.e. $\left(a \overset{SE}{\approx} b \wedge b \overset{SE}{\approx} c\right) \Rightarrow a \overset{SE}{\approx} c$. Recall that there may be several TFI cones for a node because of the existence of multiple choice nodes. Thus the proof must be valid in this setting. Let $x = \{x_i\}$ be the nodes of the common cut used in rewriting the nodes *a* and *b*, and let $y = \{y_i\}$ be the cut nodes of the common cut for *b* and *c*. Since these are both cuts for *b*, their union, $x \cup y$, is also a cut for *b*. Such a cut may be redundant, with some of the nodes in the cut having a cut contained entirely in $x \cup y$. Let $\{x^1, y^2\}$ be a minimal subset in that it does not contain a node with this property. Clearly, $\{x^1, y^2\}$ is a common cut for $\{a, c\}$. The remaining nodes, $x^2 \in x$ and $y^1 \in y$ each have a cut contained in $\{x^1, y^2\}$.

Now we use the notation $\{x_d^1, x_d^2\}$ to denote the *sequential* cut variables of the cut x and similarly for $\{y_d^1, y_d^2\}$. The subscript denotes that some of the variables may be duplicated with different time indices. By construction, there exist functions $y_d^1 = G(x_d^1, y_d^2)$ and $x_d^2 = H(x_d^1, y_d^2)$. Since

$$f_a(x_d^1, x_d^2) = f_b(x_d^1, x_d^2) \text{ and } f_b(y_d^1, y_d^2) = f_c(y_d^1, y_d^2),$$

then

$$\tilde{f}_a(x_d^1, y_d^2) \equiv f_a(x_d^1, H(x_d^1, y_d^2)) = f_b(x_d^1, H(x_d^1, y_d^2)) \equiv \tilde{f}_a(x_d^1, y_d^2),$$
$$f_b(G(x_d^1, y_d^2), y_d^2) = f_c(G(x_d^1, y_d^2), y_d^2) \equiv \tilde{f}_c(x_d^1, y_d^2)$$

Hence, $\tilde{f}_a(x_d^1, y_d^2) = \tilde{f}_c(x_d^1, y_d^2)$ and therefore $a \overset{SE}{\approx} c$. **QED**

Theorem 5.1 allows us to rewrite a node multiple times and put all the results in a single choice node without the need to prove pair-wise equivalences. Thus, all choice nodes, including both combinational and sequential, can be treated uniformly and in fact in a choice node, one rewriting may be combinational and another sequential. There is no need to distinguish these different types of rewritings and there is only one type of choice node.

5.2 An Inductive Invariant

Consider the circuit *H,* obtained by replacing each choice node in the HAIG by an OR of the nodes in its choice class. The fanouts of the OR node are just the union of all fanouts of the nodes in the choice class. We denote this as $H = OR(HAIG)$. Since it has no choice nodes, *H* is a regular sequential circuit. The invariant function is defined as

$$I = \prod_w I_w = \prod_w \exists_{\pi_w} \prod_i [r_i^w = f_i^w(\pi_w)],$$

where $I_w \equiv 1$ for any combinational rewriting step *w*.

Theorem 5.2: *I* is an inductive invariant of *H*.

Proof: Assume that *I* holds at time *j*. Then I_w holds also, which by Theorem 4.1, implies that $n_w = \tilde{n}_w$ for all *w*. Let C^w denote the sub-circuit associated with rewriting step *w*. It would differ from an associated sub-circuit C_H^w of *H* only in that there might be additional OR gates embedded in C_H^w. However, by Theorem 4.1, each OR gates behaves like a buffer if *I* holds. Then C_H^w and C^w have the same behavior, which by Theorem 4.2, I_w and hence *I* is an inductive invariant of *H*. **QED**

Let E_C denote that all equivalences from combinational rewriting steps hold.

Theorem 5.3: Invariant *I* and $E_C = 1$ imply that the POs of *D* are equal to the POs of *C*.

Proof: Since all combinational equivalences always hold, and by Theorem 4.1, *I* implies that all sequential equivalences hold, then all nodes in any choice class have equal values. Since all POs are common to both *D* and *C,* these equivalences include the equivalence of any PO of *D* to that of the corresponding one of *C*. **QED**

Note that *I* is a relation among the state variables in the HAIG. Denote the latches of the original machine *D* by *r*, the latches in the final machine *C* by *s,* and the remaining latches by *t*. The invariant is written as $I(r,s,t)$. The function $J(r,s) = \exists_t I((r,s,t)$ is a binary relation between states in *D* and *C,* and it can be proved that for any pair of states $(q, \tilde{q})$, $J(q, \tilde{q}) \Rightarrow q \equiv \tilde{q}$, i.e. if $J(q, \tilde{q}) = 1$, q and $\tilde{q}$ are sequentially equivalent. Further, for any state q in the cyclic core of *D*, there exists a state $\tilde{q}$ in the cyclic core of *C* such that $J(q, \tilde{q}) = 1$, and vice-versa.

6 Verification Process

6.1 Problem Formulation

In this section, we are concerned with how a user can verify that two FSMs are equivalent. The problem is stated as follows: Given two machines, *D'* and *C'*, and a HAIG reflecting the process of synthesizing *D'* into *C'*, verify their delayed sequential equivalence. It is assumed that the latches of *D'* and *C'* appearing in the HAIG can be matched by name.

We propose two methods for solving this problem. Both methods begin by identifying the AIG subgraphs, *D* and *C* in the HAIG that correspond to *D'* and *C'*, Next *D* and *C* are proved equivalent to *D*' and *C*' using combinational equivalence checking .

The first method consists of showing that machines *D* and *C* in the HAIG are connected by a sequence of valid sequential rewritings, where each pair of the roots is delayed sequentially equivalent. This method is more efficient since after proving the two machines are equivalent to sub-machines in the HAIG, the remaining task is to re-derive the rewriting steps from the HAIG and prove they are SE. Since the cut size for each rewriting is limited (typically 5-8 variables), truth tables can be used, leading to an efficient computation whose complexity is linear in the number of rewriting steps.

Although this method is valid, a user may object that it is too closely related to the synthesis process and is simply validating each step in sequence. Therefore in this paper, we only discuss the second method, which has the psychological advantage of using a construction independent of the synthesis process. The proof consists of verifying that the function *I* of Section 5.2 is an inductive invariant of *H*, and that *I* implies that all choice node equivalences hold.

6.2 Proof by Inductive Invariant

The set of all equivalences of the root nodes from sequential rewriting will be denoted by $E_S = 1$ and those from combinational rewritings will be denoted by $E_C = 1$.

The three steps needed for proving equivalence of the two machines are:

1. Identify machines *D* and *C* in the HAIG and prove that they are combinationally equivalent to *D'* and *C*' respectively.
2. Prove that $I \Rightarrow (E_C = 1) \wedge (E_S = 1)$. This is a relatively simple SAT proof because of the large number of internal signal correspondences that can be found[6].
3. Form *H* from the HAIG as described in Section 5. Prove that *I* is an inductive invariant of *H.* This requires only one SAT instance of *H,* since the next state signals are the inputs of the latches of the same time frame. For the latch outputs, we impose that $I = 1$ and prove $I = 1$ for the latch inputs.

This constitutes a proof of delayed equivalence between *D*' and *C*' because,

1. they are combinationally equivalent to *D* and *C,* respectively;
2. *D* and *C* are contained in the HAIG;
3. whenever *I* holds in *H,* it holds forever;
4. whenever *I* holds, all nodes in a choice class (or OR node of *H*) have equal values;
5. in particular, when *I* holds, the POs of *D* are equal to the POs of *C* from then on.

It is informative to note that for Step 1 to be valid, it is necessary that both *D* and *C* are connected to each of the POs. These connections must be through a set of choice nodes because there can be only one signal driving each PO. Step 2 allows that all the choice nodes can be connected with a common OR gate and therefore allows the use of *H* instead of the HAIG in Step 3.

[6] Of course, $E_C = 1$ could be proved separately without assuming *I,* but we choose to prove all equivalences in one step. $E_C = 1$ must be proved since we need to prove that there was not a bug in the combinational rewriting steps.

The last step of proving sequential equivalence deals with the initial conditions.

7 Initialization

In practice, proving delayed sequential equivalence is not enough because any machine must be put into a known state before useful computation can be done. If synthesis involves retiming, in general it is not possible to derive an initial state for the final circuit *C* to make it sequentially equivalent to a given initial state of the original circuit. Unless some retiming moves (some forward moves) are disallowed, it is known that only after cycling the machines for a certain number of clock cycles can one expect to find equivalent initial states [20].

Two types of initialization are commonly used. One is that an initial state is given for the initial machine, and the other is that an initialization sequence is given. Cogent arguments on why the first type is relevant for industrial practice are given in [2]. In this paper, we limit our discussion to this case.

We produce a state $s=(q,\sigma,\tilde{q})$ of the HAIG, in which invariant *I* holds.

Theorem 7.1: If $s=(q,\sigma,\tilde{q})$ is a state of the HAIG at which *I* holds, then $q\equiv\tilde{q}$.

Proof: By Theorem 5.2, *I* is an invariant and by Theorem 5.3, *I* and $E_C\equiv 1$ imply that the outputs of *D* are equal to those of *C*. Since *I* holds for state *s*, then starting from state *s*, *I* holds forever. Thus the outputs of *D* and *C* are equal forever, implying $q\equiv\tilde{q}$. **QED**

We will derive an initial state $\tilde{q}$ for *C*', which is equivalent to *q*. In order to do this, we may have to disallow a rewriting step that does not satisfy an accumulated invariant. Initially, this invariant is simply a statement that the initial state is *q*.

Denote $I^{w-1}\equiv\prod_{t<w}I_t$ and $P_w=(r_0=q)I^w$ where r_0 are the latches of *D*. For a rewriting step, the invariant, I_w, is a relation among the latches lying between the cut and the two root nodes. Partition the latches, r_w, of I_w into (r_w^o,r_w^n), where r_w^o are those of the old working AIG, and r_w^n are the newly created ones. The latches r_w^o are part of the set R_{w-1} of all latches that existed up to the time of rewriting step *w*. The initial values of these must satisfy a relation $P_{w-1}(R_{w-1})=1$, which has been derived from previous rewritings. We are concerned with the existence of a "compatible" set of values ρ_w^n for the latches r_w^n. If $P_w(R_w)\equiv 0$, then this rewriting step is not allowed. Otherwise, a set of consistent values exists and we can use one as an initial value for r_w^n. At the last rewriting step ω, we choose one assignment $(q,\sigma,\tilde{q})$, where $\tilde{q}$ corresponds to the latches of *C*, such that $P_\omega(q,\sigma,\tilde{q})=1$. Since $P_\omega(R_w)=(r_0=q)I(R_\omega)$, then $I(q,\sigma,\tilde{q})=1$.

Now consider the HAIG whose latches have the initial values $(q,\sigma,\tilde{q})$. For this state, $I=\prod_w I_w=1$ holds. By Theorem 7.1, $q\equiv\tilde{q}$ and thus $\tilde{q}$ is an equivalent state for the synthesized machine.

Theorem 7.2: If *q* is in the cyclic core of *D*', then $P_w(R_w)$ is always satisfiable.

Proof: Suppose, at rewriting step *w*, $P_w(R_w)\equiv 0$, but $P_{w-1}(R_{w-1})$ is satisfiable. Let *s* be a satisfying assignment of this. Let H_{w-1} be the OR version of the HAIG up to this point. Then I^{w-1} is an inductive invariant of H_{w-1}. A rewriting step can only create or destroy dangling states and merge or split immediate equivalent states [12]. Since *s* is in the cyclic core, there exists an input sequence S_C which returns to *s* starting from *s*. Construct the circuit where the new root node $\tilde{n}$ (with no fanout) and new fanin nodes of rewriting step w are appended to H_{w-1}. Denote the resulting circuit *Q*. Apply S_C to *Q* with initial state *s* in the latches of H_{w-1}

and an arbitrary state in the new latches r_w^n. After a few iterations of S_C, any initial dangling state created by w will be replaced by a state (s,ρ_w^n) in the cyclic core of Q. Now (s,ρ_w^n) will satisfy I_w, since there is a common cut in Q which propagates its values to the latches r_w^n (Theorem 4.3). Since s satisfies P_{w-1} and (s,ρ_w^n) satisfies I_w, then (s,ρ_w^n) satisfies P_w, contradicting the assumption that $\mathrm{P}_w(R_w) \equiv 0$. **QED**

Theorem 7.2 tells us that if the given initial state q is in the cyclic core, then no rewriting step has to be disallowed. If q is not in the cyclic core, then we can reason that the rewriting steps that have to be thrown out are only those that destroy the chain of dangling states from q leading to the cyclic core. Although this requires examining if P_w is satisfiable at each step, there are two methods for overcoming the complexity of this.

1. Prove that q is in the cyclic core. This can be done by bounded model checking, i.e. unrolling D k cycles and asking for an input sequence of length k or less which drives D from q to q.
2. If this fails, then at each rewriting step, we can try to extend the current satisfying assignment, s, of P_{w-1} to P_w, which is just a matter of satisfying I_w where some latch values are restricted to agree with s. If this can't be done, we can compute a satisfying assignment of P_w and continue. If P_w is not satisfiable, rewriting step w is thrown out (see example in Fig. 7.1). Although we expect that this last step will be rare, another option is to throw out rewriting step w if the current satisfying assignment can't be extended to satisfy P_w.

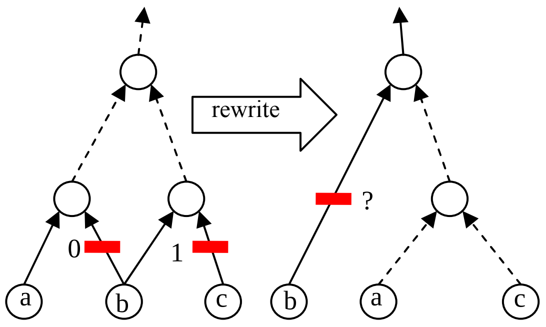

Figure 7.1.The initial state of the circuit on the left is given as 0,1, which is not in the cyclic core. No equivalent state exists for the right-most circuit; in the first time frame, the output on the left is c, but on the right, the output is 0 if the latch is 0 and $a+c$ if the latch is 1.

In summary, we can obtain a state $s=(q,\sigma,\tilde{q})$, in which I holds, by finding a satisfying assignment for I where $r_0 = q$. If q is not in the cyclic core, $s=(q,\sigma,\tilde{q})$ may not exist unless some synthesis steps are avoided.

8 Experimental Results

The presented algorithms are being implemented in a public-domain logic synthesis and verification system, ABC [3]. The experimental results reported in this paper have two goals: (1) demonstrating that HAIG construction is practical in terms of memory and runtime and (2) showing that recording the rewriting invariant for the HAIG can be done in a timely manner. The benchmarks are several public-domain industrial circuits distributed as part of Altera's QUIP package [10]. The synthesis script used in this experiment performed three iterations of sequential AIG rewriting (*irws; irws; irws –z*). The first two iterations of AIG rewriting performed replacements only if there was an improvement in the number of AIG nodes; the last iteration also performed zero-cost replacements.

Table 1. Experimental results.

Benchmark name	Synthesis							Verification		
	AIG nodes			Rewriting steps		Runtime, sec		CNF clauses		Time, sec
	D	C	HAIG	Total	Seq	Synth	HAIG	HAIG	Invar	
oc_aquarius	25056	20091	45459	26976	9732	3.55	3.77	63767	40426	0.95
oc_cordic_p2r	11846	8432	22216	8646	2848	1.35	1.37	31335	13918	0.43
oc_fpu	24916	16116	48302	21798	381	2.70	2.79	56815	1004	0.86
oc_video_jpeg	56601	43021	115055	46187	9495	7.15	7.40	153697	36380	6.95
oc_video_dct	46432	32578	87145	40667	17320	6.33	6.52	111036	56406	1.53
Ratio	1.00	0.72	1.91	1.00	0.27	1.00	1.04	1.00	0.37	

8.1 HAIG Construction during Synthesis

Table 1 shows the benchmark names and number of AIG nodes present in the original and final designs, and in the HAIG. In our current implementation, each AIG node with the fanout information uses about 64 bytes of RAM. Next, the table lists the total number of rewriting steps and the number of sequential rewriting steps. A rewriting step is considered sequential if at least one latch was moved by backward retiming to the boundary of common cut. The synthesis part of the table also lists the runtime of sequential synthesis without HAIG construction and with HAIG construction. The runtimes are reported on a 1.6GHz Intel CPU.

8.2 Exploiting HAIG during Verification

The verification section of Table 1 shows the number of clauses after converting the HAIG into the CNF form using an improved circuit-to-CNF conversion [21]. The table also gives the number of additional CNF clauses needed to represent the rewriting invariant for each sequential rewriting step. The runtime of deriving the invariant is given in the last column.

The table does not contain the runtime of the SAT solver needed to prove the invariant because this part of the project is not yet finished.

8.3 Discussion

The experimental results indicate that the HAIG has a relatively small size because of substantial node sharing in it. Since HAIG is an AIG, highly compact binary file formats, such as AIGER [19], can be used to store it on disk as an "equivalence certificate" for later use by a verification system. The results confirm that the HAIG construction is practical for large sequential designs subjected to sequential synthesis.

9 Conclusions and Future Work

This paper presented a method for sequential verification of two FSMs where one was transformed into the other by a sequence of sequential rewritings. This is done by efficiently recording all nodes that occurred during the synthesis process, resulting in a history AIG (HAIG). A HAIG can also be used to benefit synthesis by allowing a technology mapper to see more versions of the circuit. For verification, an inductive invariant of the HAIG can be used to prove delayed sequential equivalence of the two FSMs and to find equivalent initial states.

We conjecture that the set of all equivalences, $E_C E_S$, at the choice nodes of the HAIG is a $k_{\max} - 1$ step invariant of the HAIG where $k_{\max}$ is the maximum of the sequential depths of the rewriting steps, i.e. after $k_{\max} - 1$ cycles, all equivalences hold thereafter. In fact, we have shown that this is true in tests on several large benchmarks. However, we have seen from experiments on some benchmark examples that $k_{\max}$ can be as much as 10 and using this method would entail that the proof of invariance would require writing $k_{\max}$ copies of the SAT instances corresponding to the HAIG.

With this mechanism, sequential verification takes on a complexity of two CEC problems in the size of the circuit and one invariance check on a single copy of the HAIG. Additionally, the many structural similarities in these SAT problems make these problems relatively easy.

Acknowledgement

This research was supported in part by SRC contracts 1361.001 and 1444.001, and by the California Micro program with industrial sponsors, Altera, Intel, Magma, and Synplicity.

References

[1] J. Baumgartner and A. Kuehlmann, "Min-area retiming on flexible circuit structures", *Proc. ICCAD '01*, pp. 176-182.

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

[3] Berkeley Logic Synthesis and Verification Group, *ABC: A system for sequential synthesis and verification*, *Release 70127*. http://www.eecs.berkeley.edu/~alanmi/abc/

[4] A. Biere, *AIGER: A format for And-Inverter Graphs*. http://fmv.jku.at/aiger/

[5] P. Bjesse and K. Claessen. "SAT-based verification without state space traversal". *Proc. FMCAD '00,* LNCS, Vol. 1954, pp. 372-389.

[6] P. Bjesse and A. Boralv, "DAG-aware circuit compression for formal verification", *Proc. ICCAD '04*, pp. 42-49.

[7] M. Case, A. Mishchenko, and R. Brayton, "Inductively finding a reachable state space over-approximation", *Proc. IWLS '06*, pp. 172-179.

[8] S. Chatterjee, A. Mishchenko, R. Brayton, X. Wang, and T. Kam, "Reducing structural bias in technology mapping", *Proc. ICCAD '05*, pp. 519-526. http://www.eecs.berkeley.edu/~alanmi/publications/2005/iccad05_map.pdf

[9] C. A. J. van Eijk, "Sequential equivalence checking based on structural similarities", *IEEE Trans. CAD*, Vol. 19(7), July 2000, pp. 814-819.

[10] M. Hutton and J. Pistorius. *Altera QUIP benchmarks*. http://www.altera.com/education/univ/research/unv-quip.html

[11] *IWLS '05 Benchmarks*. http://iwls.org/iwls2005/benchmarks.html

[12] J-H. R. Jiang and R. Brayton, "Retiming and resynthesis: A complexity perspective", *IEEE Trans. CAD,* Vol. 25(12), Dec 2006, pp. 2674-2686. http://www.eecs.berkeley.edu/~brayton/publications/2006/tcad06_r&r.pdf

[13] C. E. Leiserson and J. B. Saxe. "Retiming synchronous circuitry", *Algorithmica*, 1991, Vol. 6, pp. 5-35.

[14] S. Malik, E. Sentovich, R. Brayton, and A. Sangiovanni-Vincentelli, "Retiming and resynthesis: Optimizing sequential networks with combinational techniques", *IEEE Trans. CAD*, Vol. 10(1), pp. 74-84, Jan. 1991.

[15] A. Mishchenko, S. Chatterjee, R. Brayton, and P. Pan, "Integrating logic synthesis, technology mapping, and retiming", *ERL Technical Report*, EECS Dept., UC Berkeley, April 2006. http://www.eecs. berkeley.edu/~alanmi/publications/ 2006/tech06_int.pdf

[16] A. Mishchenko, S. Chatterjee, and R. Brayton, "DAG-aware AIG rewriting: A fresh look at combinational logic synthesis", *Proc. DAC '06*, pp. 532-536. http://www.eecs.berkeley.edu/~alanmi/publications/ 2006/dac06_rwr.pdf

[17] A. Mishchenko and R. Brayton, "Sequential rewriting", to appear.

[18] M. Mneimneh and K. Sakallah, "REVERSE: Efficient sequential verification for retiming", *Proc. IWLS '03*, pp. 133-139.

[19] P. Pan and C.-C. Lin, "A new retiming-based technology mapping algorithm for LUT-based FPGAs," *Proc. FPGA '98*, pp. 35-42.

[20] N. Shenoy, K. Singh, R. Brayton, and A. Sangiovanni-Vincentelli, "On the temporal equivalence of sequential circuits", *Proc. DAC '92*, pp. 405-409.

[21] M. N. Velev. "Efficient translation of Boolean formulas to CNF in formal verification of microprocessors", *Proc. ASP-DAC '04*, pp. 310-315.