Principles of Sequential-Equivalence Verification

Maher N. Mneimneh and Karem A. Sakallah
University of Michigan

Editor's note:
This article is a general survey of conceptual and algorithmic approaches to sequential-equivalence checking. Although this fundamental problem is very complex, recent advances in satisfiability solvers and ATPG approaches are making good headway.

—Carl Pixley, Synopsys

CHECKING WHETHER two models of a design are functionally equivalent is a crucial step in a transformation-based design flow, in which a designer obtains lower-level implementation models by manual or automatic translation from higher-level specification models. In a manual translation, the designer constructs a transistor- or gate-level model of a register-transfer-level (RTL) specification. In an automatic translation, synthesis tools transform the RTL model into a gate-level model. Usually, designers use automatic translation to synthesize a design's control portion and manual translation to synthesize its data path components.

In either case, the process is not guaranteed to be error free. For example, a designer might erroneously interpret the RTL description and generate a nonequivalent implementation. With synthesis tools, problems are less frequent but are not completely eliminated. Although the algorithms used for synthesis are presumed to be correct by construction, errors might exist in their software implementations. Because of the sheer size of such implementations, applying software verification to check their correctness is currently beyond the scope of most tools. In addition to software errors, manual changes, usually applied after automatic synthesis to improve a design parameter, might introduce errors in the implementation models.

Traditionally, designers use simulation to check the functional equivalence of specification and implementation models. Although simulation can eliminate some or most design errors, it can never completely certify design correctness. Formal equivalence verification (FEV) is a viable alternative that has gained wide industrial acceptance. FEV, which uses automata theory and mathematical logic to formally reason about the correctness of design transformations, is broadly divided into two categories: combinational and sequential. Combinational-equivalence verification (CEV) is appropriate when the transformations applied affect the circuit's combinational logic only. Examples of such optimizations are two-level and multilevel minimization, timing optimization, and technology mapping. CEV has become mainstream in today's design flows, and current CEV tools can handle fairly complex designs.

Designers use sequential-equivalence verification (SEV) for transformations that don't preserve the correspondence between memory elements in both circuits. Examples of such transformations include retiming,[1] state minimization,[2] state encoding,[2] sequential-redundancy removal, and redundancy addition and removal. Despite considerable research, SEV tools and algorithms suffer from an inherent, widely acknowledged state explosion problem. That is, for a given design, the number of potential states is exponential in the number of state devices. Checking equivalence requires, in one form or another, exploration of the design's state space. Thus, this problem has exponential complexity. As a result, designers rarely use sequential optimizations despite their potential improvements of design parameters such as area, speed, and power.

In recent years, researchers have proposed new

0740-7475/05/$20.00 © 2005 IEEE Copublished by the IEEE CS and the IEEE CASS

approaches to tackling the limitations of the basic SEV algorithms. Although the problem is very hard from a complexity theory perspective, many heuristic improvements have worked well in practice. Among recent advances are symbolic traversal, induction, and structural approaches. In this article, we review SEV solutions ranging from basic algorithms to more scalable approaches. In addition to efficient algorithms, much research has dealt with understanding the meaning of sequential equivalence, resulting in various notions of equivalence. We summarize these notions and their interrelationships.

Preliminaries

Basic SEV concepts include sequential circuits, state equivalence, synchronizing and initializing sequences, sequential equivalence, and product machines.

Sequential-circuit representation

Figure 1a shows the structure of a synchronous sequential circuit. The circuit has a finite number m of inputs $(x_1, x_2, \ldots, x_m)$, a finite number l of outputs $(z_1, z_2, \ldots, z_l)$, and a finite number n of state or memory elements $(y_1, y_2, \ldots, y_n)$. The circuit's combinational part consists of k internal signals $(w_1, w_2, \ldots, w_k)$ representing the outputs of combinational gates. Each of these signals takes one of two possible values, 0 or 1. Clock signal *clk* synchronizes the memory elements' operation. We assume that all the memory elements are edge sensitive and change their values on the rising edge of the clock. We refer to these memory elements as flip-flops.

Figure 1b shows an example of a sequential circuit. We model sequential circuits using finite-state machines (FSMs). A Mealy FSM M is defined as a sextuple: $M = (Q, \Sigma, \Delta, \delta, \lambda, Q^0)$, where Q is a finite set of states, Σ is the input alphabet, Δ is the output alphabet, $\delta: Q \times \Sigma \to Q$ is the state-transition function, $\lambda: Q \times \Sigma \to \Delta$ is the output function, and Q^0 is the set of initial states. In modeling the circuit in Figure 1b with an FSM, we have $Q = \{00, 01, 10, 11\}$, $\Sigma = \{0, 1\}$, and $\Delta = \{0, 1\}$. Denoting the next-state value of state variable y as y^+, we have $y_1^+ = x_1'y_2 + x_1y_1y_2'$ and $y_2^+ = x_1y_1' + x_1y_2$ representing the next-state equations. The next-state function is $(y_1^+, y_2^+) = \delta(y_1, y_2, x_1) = (x_1'y_2 + x_1y_1y_2', x_1y_1' + x_1y_2)$. We define the transition relation $T(y_1^+, y_2^+, y_1, y_2, x_1)$ as the conjunction of all the next-state equations: $T(y_1^+, y_2^+, y_1, y_2, x_1) = (y_1^+ = x_1'y_2 + x_1y_1y_2')(y_2^+ = x_1y_1' + x_1y_2)$. Alternatively, we can define the transition relation $T(y_1^+, y_2^+, y_1, y_2)$ on present and next-state variables only by existential elimination of the input variables from $T(y_1^+, y_2^+, y_1, y_2, x_1)$: $T(y_1^+, y_2^+, y_1, y_2) = \exists x_1\, T(y_1^+, y_2^+, y_1, y_2, x_1)$. The output function is $\lambda(y_1, y_2, x_1) = z_1 = x_1y_1y_2 + x_1y_1'y_2' + x_1'y_1y_2'$.

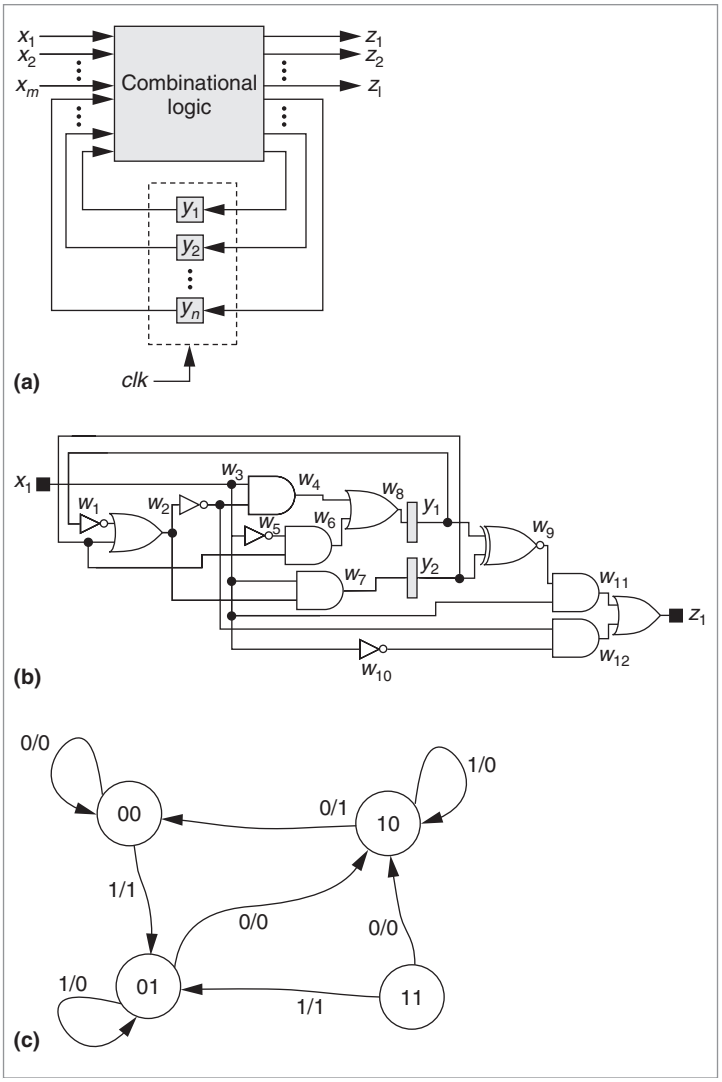


Figure 1. Huffman model of a synchronous sequential circuit (a), example sequential circuit (b), and its corresponding state transition graph (c).

An FSM is represented visually by a state transition graph. STG(M), the state transition graph of FSM M, is a labeled directed graph $\langle V, E \rangle$ in which each vertex $v \in V$ corresponds to state s_i of M (and is labeled s_i), and each edge $e \in E$ between two vertices labeled s_i and s_j corresponds to a transition from state s_i to state s_j in M. The edge is labeled i_k/o_l, where i_k is the input that causes the transition from s_i to s_j and o_l is the output during that transition. Figure 1c shows the STG of the circuit in Figure 1b.

We denote by x^i $(i \geq 0)$ the input in clock cycle i to the FSM where $x = (x_1, x_2, \ldots, x_m)$, and by $x^0 \cdot x^1 \cdot \ldots \cdot x^{u-1}$ a sequence of u consecutive inputs. If applying this input

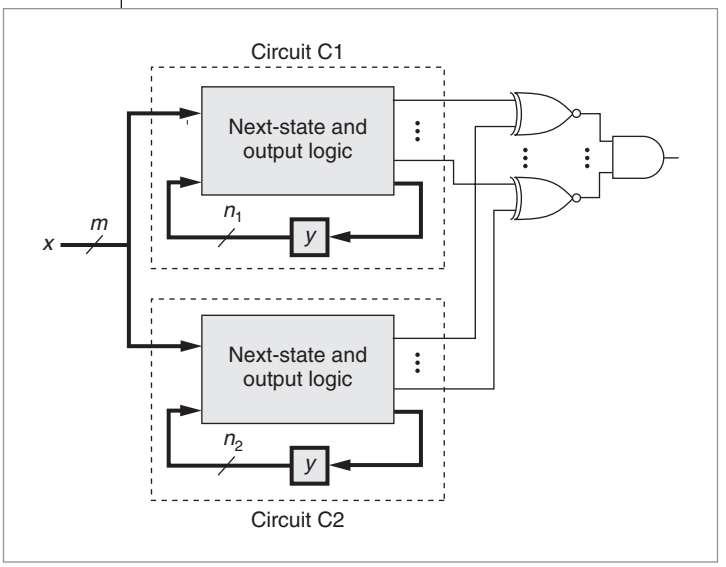


Figure 2. Product machine (miter) of two sequential circuits.

sequence from state s_0 causes the machine to transition from state s_{i-1} to state s_i on input x^{i-1}, we write $s_0 \xrightarrow{x^0} s_1 \xrightarrow{x^1} s_2 \ldots s_{u-1} \xrightarrow{x^{u-1}} s_u$. If the input sequence is irrelevant, we write $s_0 \rightarrow s_1 \rightarrow s_2 \ldots s_{u-1} \rightarrow s_u$.

We can generalize the modeling just described as circuits containing multiple clock domains, gated clocks, and level-sensitive memory elements (or latches).

State equivalence

Two states s_1 and s_2 of machine M are equivalent ($s_1 \sim s_2$) if for every possible input sequence applied from these states, the same output sequence results. If s_1 and s_2 are not equivalent, they are distinguishable. Any input sequence that produces different outputs is called a distinguishing sequence for these two states. In Figure 1c, 00 and 10 are not equivalent because they produce different outputs when the input is 1.

Synchronizing and initializing sequences

A synchronizing sequence of machine M is an input sequence that drives M to a specific state s_{synch} when applied from any state of M; s_{synch} is a synchronizing state of M. M is synchronizable if it has at least one synchronizing sequence. In Figure 1d, input sequence $0 \cdot 0$ is a synchronizing sequence because applying it from any state takes the machine to state 00.

An initializing sequence is a synchronizing sequence identifiable through three-valued logic simulation. In three-valued logic simulation, a signal in the circuit takes a value from the set $\{0, 1, X\}$, where X represents the unknown value. Because three-valued logic simulation suffers from information loss, a synchronizing sequence is not necessarily an initializing sequence. If we denote the set of synchronizing sequences and the set of initializing sequences of machine M as S_M and I_M, $I_M \subseteq S_M$.

Sequential equivalence

We can check whether two circuits are sequentially equivalent by applying the classical notion of FSM equivalence.[2,3] Classical FSM equivalence requires each state in one FSM to be equivalent to a state in the other and vice versa. However, this requirement may be too stringent for some sequential circuits. Usually, a sequential circuit has an input sequence, called the reset sequence, that is applied when circuit operation starts. The reset sequence forces the value of some or all memory elements to 0 or 1 and consequently limits the possible initial states. When the reset operation brings the circuit to a single initial state, the classical notion reduces to proving the initial states' equivalence. For some circuits (for example, data path circuits), many flip-flops can remain uninitialized after reset. For others, the reset sequence might not be available until very late in the design cycle. For these cases, various equivalence notions have been devised.

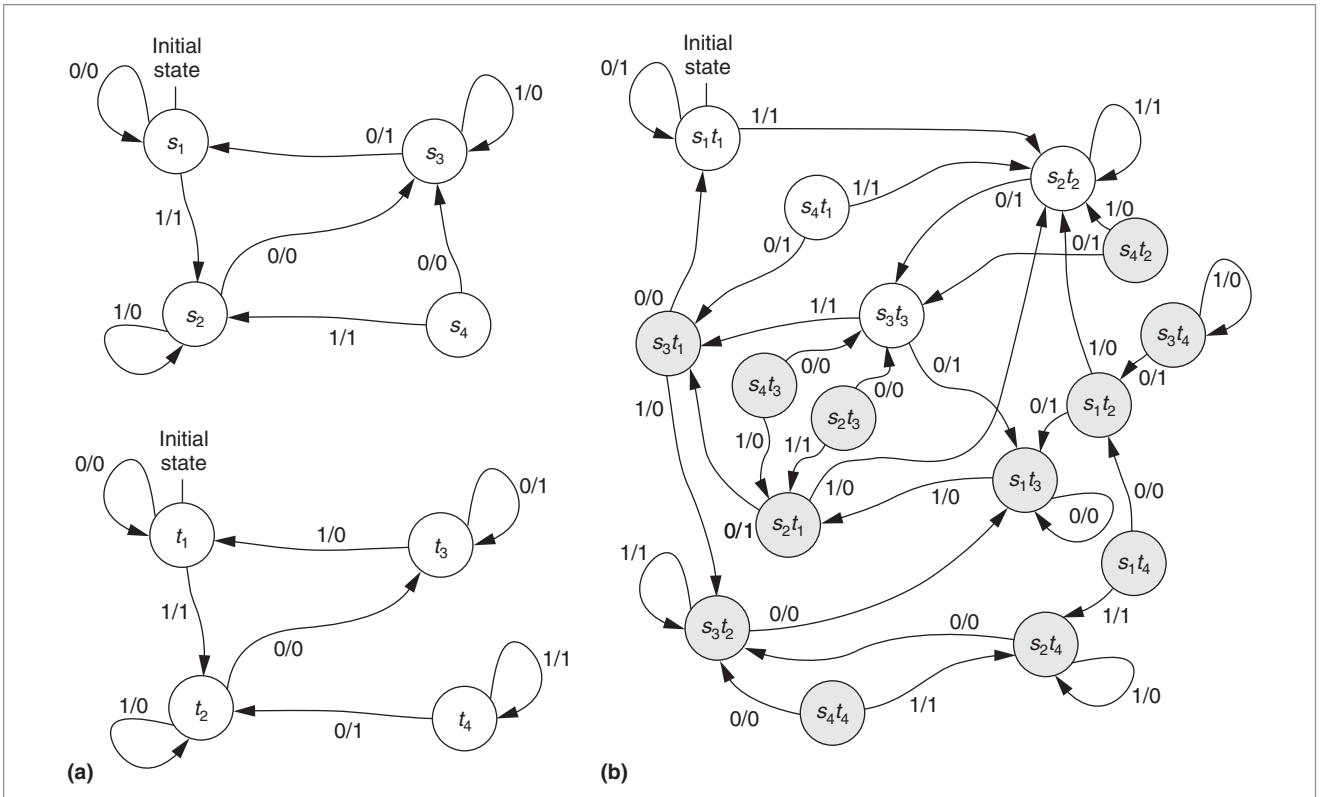
Product machine

To check for equivalence, we must search for an input sequence that produces different outputs. We can do this by connecting together both machines' inputs, connecting the corresponding outputs to an exclusive NOR (XNOR) gate's inputs, and connecting the XNOR gate's outputs to an AND gate. The resulting circuit is usually called a miter circuit, and its FSM is called the product machine. Figure 2 shows the product machine of two sequential circuits. Each state of the product machine corresponds to a state pair s_1s_2, where s_1 is a state of M_1 and s_2 is a state of M_2. We call a state of the product machine with output 0 unsafe, and a state with output 1 safe. State s_j is reachable from s_i if there is an input sequence that takes the machine from state s_i to s_j or if $i = j$.

Two states s_1 of M_1 and s_2 of M_2 are equivalent if all states reachable from state s_1s_2 of the product machine are safe. Figure 3a shows two FSMs, and Figure 3b shows their product machine. A pointer indicates that a state is an initial state. Unsafe states of the product machine are shaded.

Circuits with one initial state

When each circuit has one initial state, verifying



Iteration i	$Reach^i$	New^i
0	$\{s_1\ t_1\}$	$\{s_1\ t_1\}$
1	$\{s_1\ t_1,\ s_2\ t_2\}$	$\{s_2\ t_2\}$
2	$\{s_1\ t_1,\ s_2\ t_2,\ s_3\ t_3\}$	$\{s_3\ t_3\}$
3	$\{s_1\ t_1,\ s_2\ t_2,\ s_3\ t_3,\ s_1\ t_3,\ s_3\ t_1\}$	$\{s_1\ t_3,\ s_3\ t_1\}$

(c)

Figure 3. Two FSMs (a), their product FSM (b), and the iterations of forward traversal on the product machine (c). Unsafe product machine states are shaded.

equivalence reduces to checking the initial states' equivalence. There are two common approaches. The first locally checks whether the two states are equivalent; the second works globally by computing the whole set of nonequivalent state pairs and checking whether the initial state pair belongs to this set.

In the first approach, we apply breadth-first search to find the set of states reachable from the product machine's initial state. We call this procedure forward FSM traversal. If all reachable states are safe, the two circuits are equivalent; otherwise, they are not. The product FSM in Figure 3b illustrates this procedure. At a given iteration, we keep track of $Reach^i$, the set of states reached at iteration i from the initial state, and New^i, the set of new states reached at iteration i. Initially, we set $Reach^0$ and New^0 to the initial state. In Figure 3, to check whether the initial states s_1 and t_1 are equivalent, we traverse the product machine starting from state s_1t_1. Figure 3c shows the traversal iterations. Iteration 3 reaches unsafe states s_1t_3 and s_3t_1. Thus, s_1 and t_1 are not equivalent, and consequently the two machines are not equivalent.

In the alternative procedure, we compute the set of nonequivalent state pairs. If the initial state pair does not belong to this set of nonequivalent state pairs, the two circuits are equivalent; otherwise, they are not. Observe that an unsafe state of the product machine corresponds to a nonequivalent state pair, and that a state that can reach an unsafe state also corresponds to a nonequivalent state pair. Thus, we start with the set of unsafe states and add their predecessors by backward

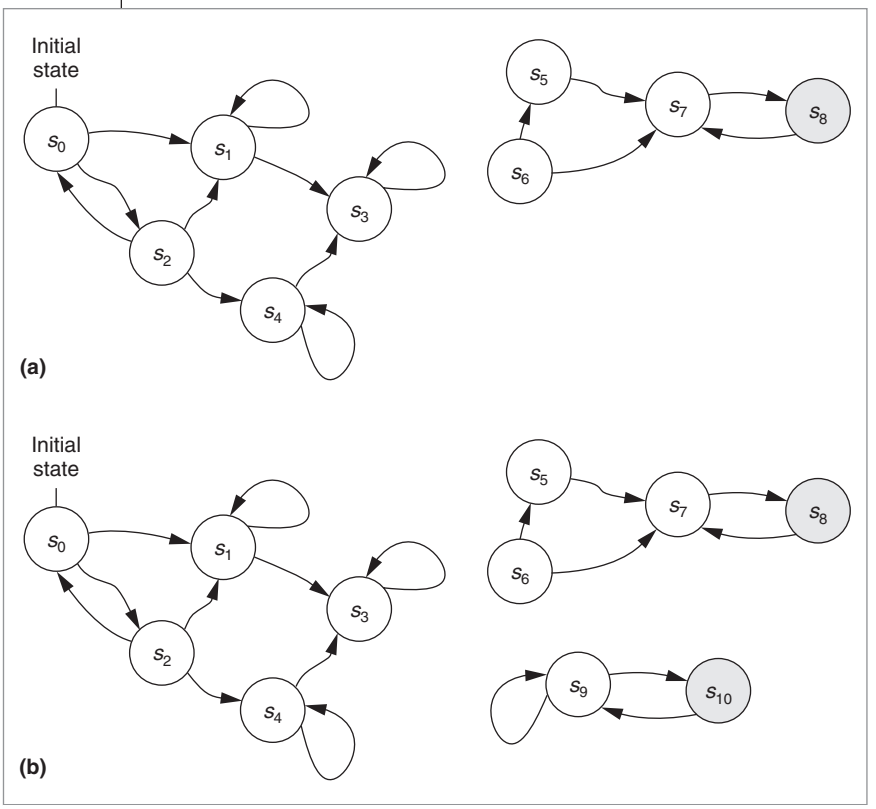


Figure 4. Examples of product machines: satisfying induction at depth 4 (a) and violating induction at all depths (b). Edge labels are omitted for simplicity. Unsafe states are shaded.

FSM traversal until we reach a fixed point. As in the previous approach, we keep track of $BReach^i$, the set of states reached at iteration i from unsafe states, and $BNew^i$, the set of new states reached at iteration i. For Figure 3, it is easy to verify that after three iterations, the initial state pair belongs to $BReach^3$. Thus, the two machines are not equivalent.

State explosion and BDD-based symbolic approaches

Reducing sequential equivalence to a graph traversal problem causes exponential complexity because the state transition graph's size is exponential in the number of memory elements of the corresponding circuit. In particular, a circuit with n memory elements results in a graph with 2^n vertices. This phenomenon is known as the "state explosion" problem. From a computational-complexity perspective, we know of no algorithms that can verify sequential equivalence in polynomial time.

Several heuristic approaches have been devised to tackle this state explosion problem. Symbolic graph traversal[4,5] is one example that has proven effective in practice. The main idea of symbolic approaches is to perform the search by manipulating the characteristic functions of sets and relations instead of enumerating states one by one. These approaches use binary decision diagrams (BDDs),[6] an efficient data structure for representing and manipulating Boolean functions. Despite the success of BDDs on various practical cases, their memory requirements can be prohibitive—known as the memory explosion problem.

CNF-based induction approaches

Induction approaches[7] are based on the following observation: If we show that the initial state is safe and that every safe state transitions to safe states only, we can conclude that all states reachable from the initial state are safe. This observation is similar to mathematical induction, in which checking whether the initial state is safe corresponds to the base step, and checking whether every safe state transitions to safe states corresponds to the inductive step. However, notice that this observation, unlike mathematical induction, makes a one-way implication. Consequently, it is sufficient but not necessary to prove equivalence, because although safe state S_{safe} might transition to an unsafe state, S_{safe} might not be reachable from the initial state. For example, transition $s_7 \rightarrow s_8$ in Figure 4a violates the inductive step (s_7 is safe but s_8 is not) even though all reachable states from the initial state are safe. When the inductive step fails, we repeat the proof by increasing the induction depth. At an induction depth of 2, the base step checks the validity of $I(s_0) \cdot T(s_0, s_1) \rightarrow P(s_0) \cdot P(s_1)$, and the inductive step checks the validity of $T(s_0, s_1) \cdot T(s_1, s_2) \cdot P(s_0) \cdot P(s_1) \rightarrow P(s_2)$, where $I(s)$ is 1 if s is an initial state and 0 otherwise, and $P(s)$ is 1 if s is safe and 0 otherwise.

In Figure 4a, transitions $s_5 \rightarrow s_7 \rightarrow s_8$ violate the inductive step at depth 2, and transitions $s_6 \rightarrow s_5 \rightarrow s_7 \rightarrow s_8$ violate it at depth 3. At depth 4, the inductive step holds. Thus, we can conclude that the product machine corresponds to equivalent circuits.

This approach cannot disprove equivalence even if we increase the induction depth indefinitely. For example, consider the transition system in Figure 4b.

Induction of any depth fails: For depth i, the transitions

$$\underbrace{s_9 \rightarrow s_9 \rightarrow \ldots \rightarrow s_9 \rightarrow s_{10}}_{i \text{ transitions}}$$

violate the inductive step. We can fix this by providing stronger versions of the inductive step. To do this, we define a walk and a path of length i of the STG:

$$walk_i(s_0, s_1, \ldots, s_i) = T(s_0, s_1) \cdot \ldots \cdot T(s_{i-1}, s_i)$$

$$path_i(s_0, s_1, \ldots, s_i) = T(s_0, s_1) \cdot \ldots \cdot T(s_{i-1}, s_i) \cdot \prod_{0 \le j < k \le i} s_j \neq s_k$$

Intuitively, $walk_i(s_0, s_1, \ldots, s_i) = 1$ if there is a walk of length i in the STG, and $path_i(s_0, s_1, \ldots, s_i) = 1$ if there is a path (no repeated states) of length i in the STG. For the complete algorithm, the base step at depth i checks the validity of

$$I(s_0) \cdot walk_i(s_0, s_1, \ldots, s_i) \rightarrow P(s_0) \cdot \ldots \cdot P(s_i)$$

and the inductive step at depth i checks the validity of

$$path_{i+1}(s_0, s_1, \ldots, s_i, s_{i+1}) \cdot P(s_0) \cdot \ldots \cdot P(s_i) \rightarrow P(s_{i+1})$$

Thus, we can prove the STG in Figure 4b safe at induction depth 4. As another example, consider the FSM of Figure 3b. For $i = 0$, base step $I(s_0) \rightarrow P(s_0)$ is valid. Inductive step $path_i(s_0, s_1) \times P(s_0) \rightarrow P(s_1)$ is not valid because s_3t_3 satisfies P, but it transitions to s_1t_3, which does not satisfy P. Next, we increase the induction depth to 1. The base step is still valid, but the inductive step is not; a counterexample is $s_2t_2 \rightarrow s_3t_3 \rightarrow s_3t_1$. When we increase the induction depth to 2, the inductive step is not valid because of the following counterexample: $s_1t_2 \rightarrow s_2t_2 \rightarrow s_3t_3 \rightarrow s_3t_1$. When we increase the induction depth to 3, the base step is not valid because of counterexample $s_1t_1 \rightarrow s_2t_2 \rightarrow s_3t_3 \rightarrow s_1t_3$. Thus, the two circuits are not equivalent.

The advantage of the induction-based approach is that we can verify both the base and inductive steps with a satisfiability solver. To do this, we convert the base and inductive steps to conjunctive normal form (CNF), thus circumventing the memory explosion problem of BDD-based symbolic approaches. However, a disadvantage is that proving a property might require a very high induction depth. To correct this, we strengthen the inductive step by replacing $path_i(s_0, s_i)$ with $shortestpath_i(s_0, s_1, \ldots, s_i)$. Intuitively, *path* searches for a path with no repeated vertices, whereas *shortestpath* searches for the longest shortest path in the transition graph. With this stronger inductive step, we can conclude that in Figure 4b, equivalence holds at depth 3 instead of depth 4. Figure 5 illustrates the trade-offs among different inductive steps. For *walk*, *path*, and *shortestpath*, we compare the required number of iterations to prove the property and the required computational resources for a given iteration. *Walk* requires the largest number of iterations but the least computational resources per iteration. *Shortestpath* requires the least number of iterations but the most computational resources per iteration. *Walk* is incomplete because it might fail to prove equivalence even with an indefinitely increased induction depth, whereas *path* and *shortestpath* are complete.

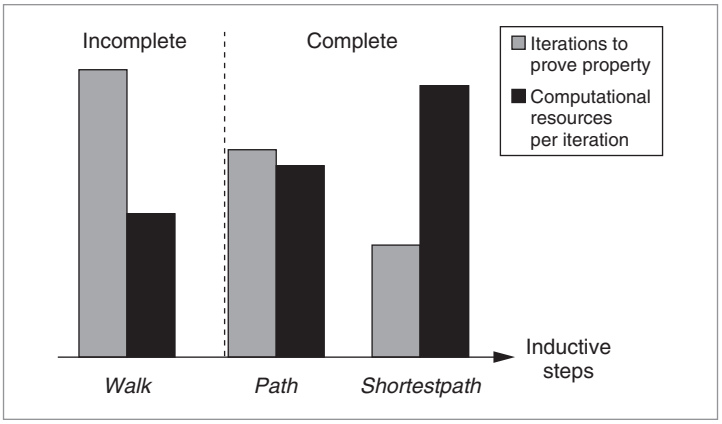
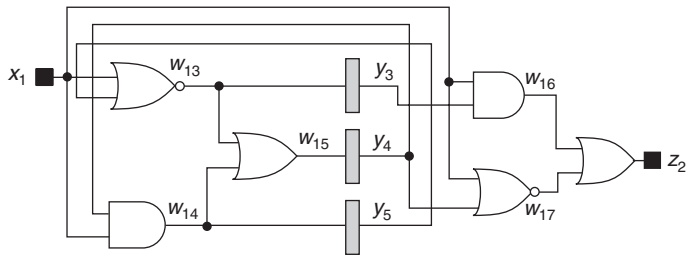


Figure 5. Trade-offs among inductive steps.

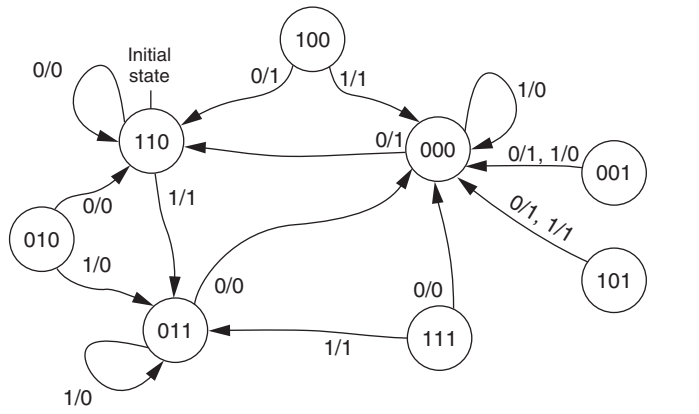
Structure-driven approaches

Structure-driven approaches use functional relations that exist among the two circuits' signals to simplify the verification task.[8-11] The observation behind these approaches is that the circuits being compared are related in some fashion, typically one being derived from the other through a sequence of transformations. A by-product of such transformations are functional relations, which, appropriately used, can make verification more tractable. In general, these relations take the form of implication, equality, or more complex Boolean functions.

As an example, consider the circuit in Figure 6a and its corresponding STG in Figure 6b. Figure 6c shows the result of applying forward traversal on the product machine formed between the circuits of Figure 1b and Figure 6a. Examining the states reachable at iterations 0, 1, and 2, we notice that state variables y_2 and y_5 are equivalent. Such equivalence relations can substantially



(a)



(b)

Iteration i	$Reach^i$
0	{(00;110)}
1	{(00;110), (01;011)}
2	{(00;110), (01;011), (10;000)}

(c)

Figure 6. Sequential circuit (a), its STG (b), and forward traversal on the product machine of the circuit in Figure 1b and Figure 6a (c).

speed up verification by eliminating the need for searching states violating the equivalence relation. As a generalization, we can also look for delayed-equivalent signals. Two signals are delayed-equivalent if the output of one is identical to that of the other but delayed by some number of clock cycles. For example, in the circuits of Figure 1b and Figure 6a, $w_{13}(t) = w_9(t+1)$ for $t \geq 0$, where $w_i(t)$ is the value of signal w_i at time t. To verify this, we notice that $w_{13}(t) = x_1'(t) \cdot y_5'(t)$ and $w_9(t+1) = y_1(t+1) \cdot y_2(t+1) + y_1'(t+1) \cdot y_2'(t+1)$. By substituting the next-state values for y_1 and y_2, we get $w_9(t+1) = x_1'(t) \cdot y_2'(t)$. But we have seen previously that $y_2(t) = y_5(t)$; thus, we conclude that $w_{13}(t) = w_9(t+1)$.

In addition to equivalences, we can use implication relations between state variables of the product machine to simplify verification. For example, examining the reachable states at iteration 2 in Figure 6c, we notice that y_1y_2 and $y_1'y_2'$ are both implicants of y_3. More-complex functional relations can be identified as well. For instance, it is easy to verify that the functional relation $y_4 = y_1' + y_2$ holds for all iterations in Figure 6c. For this relation, we call y_4 a functionally dependent variable since its value can be derived from other variables (y_1 and y_2 in this case). Functionally dependent variables are one cause of BDD blowups. Detecting and removing these variables during forward traversal can enhance the basic algorithm.

Circuits with an unknown initial state

Because additional transistors are needed to implement the reset operation, a reset flip-flop has a larger area than a nonreset flip-flop. Furthermore, routing a global reset signal to every flip-flop introduces additional area overhead. We can avoid this overhead by using nonreset flip-flops. With such flip-flops, the assumption of a single initial state no longer holds. To remedy this, researchers have proposed various notions of sequential equivalence for circuits with an unknown initial state.

Classical FSM equivalence

Under the classical notion of FSM equivalence, two machines are equivalent if for each state in one machine, there is a corresponding equivalent state in the other and vice versa.[2,3] Using this definition, it is easy to show that the two machines M_1 and M_2 in Figure 7 are equivalent. Classical FSM equivalence makes no assumptions about the circuit's operation. In the presence of a reset sequence, some states might never occur during operation, and thus this notion becomes too restrictive.

Sequential hardware equivalence

Sequential hardware equivalence (SHE) is a theory of sequential equivalence for circuits with no known initial state.[12] Under SHE, two machines are equivalent if, regardless of what power-up states they are in, they can be driven to two equivalent states after which their behavior will be identical. The input sequence that brings both circuits to equivalent states is called an aligning sequence. The two machines need not produce identical outputs during the aligning process (that is, their transient behavior can be arbitrary.) We can show that two machines are equivalent if and only if there exists a single aligning sequence that aligns all state pairs to some equivalent state pair. In addition, if the two machines having equivalent states are syn-

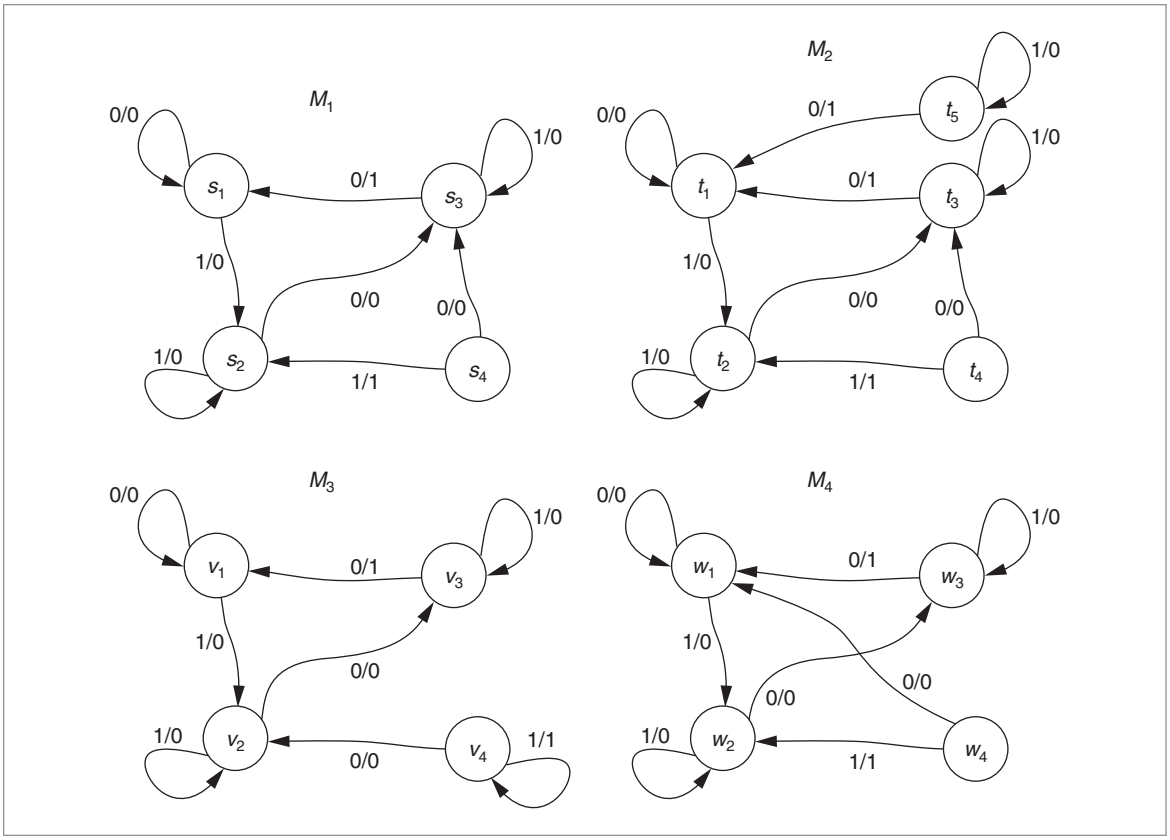


Figure 7. Examples of machine equivalence notions. $M_1 \approx_{fsm} M_2$ because $t_1 \sim s_1$, $t_2 \sim s_2$, $t_3 \sim s_3$, $t_4 \sim s_4$, and $t_5 \sim s_3$. $M_1 \approx_{she} M_3$ because input sequence $0 \cdot 0 \cdot 0$ aligns all state pairs to equivalent pair (s_1, v_1). $M_4 \leq_{safe} M_1$ because for all input sequences, states w_1, w_2, and w_3 of M_4 behave like states s_1, s_2, and s_3 of M_1. State w_4 behaves like state s_1 for all input sequences starting with a 0 and like state s_4 for all input sequences starting with a 1. Similarly, $M_1 \leq_{safe} M_4$ and thus $M_1 \approx_{safe} M_4$.

chronizable, we can form one possible universal aligning sequence by concatenating a synchronizing sequence of one machine with that of the other.

Machines M_1 and M_3 of Figure 7 are equivalent under SHE. Note that SHE imposes no restrictions on the synchronizing sequences of the two machines checked for equivalence. This is problematic when the circuit's environment can generate some sequences but not others. For example, the environment of M_1 might not be able to produce M_3's synchronizing sequence $0 \cdot 0 \cdot 0$ because it allows only two cycles for initialization. Consequently, replacing M_1 with M_3 might result in unexpected behavior.

Practically, SHE can be very effective in checking sequential equivalence in the absence of a reset sequence. If the two designs are equivalent under SHE, we know that they are equivalent in their steady-state behavior. Once the reset sequence is available, we can use a stronger notion of equivalence based on the assumptions made about circuit operation. In that case, SHE acts as a necessary (but not sufficient) equivalence requirement.

Safe-replacement equivalence

Safe-replacement equivalence is the least distinguishing equivalence notion that makes no assumptions about a circuit's operation.[13] Machine M_2 is a safe replacement for machine M_1 (written $M_2 \leq_{safe} M_1$) if and only if for any state s_2 of M_2 and for any input sequence $\pi \in \Sigma^k$ ($|\pi| = k$), there exists some state s_1 of M_1 that produces the same output as s_2 when input sequence π is applied. Machines M_1 and M_2 are safe-replacement equivalent ($M_1 \approx_{safe} M_2$) if $M_1 \leq_{safe} M_2$ and $M_2 \leq_{safe} M_1$.

The preceding definition states that M_2 is safe-replacement equivalent to M_1 as long as the I/O behavior of every state of M_2 (M_1) can be reproduced by some state of M_1 (M_2). Note that this definition allows a state of M_2 to behave like some state of M_1 for input sequence π and like some other state of M_1 for input sequence π'.

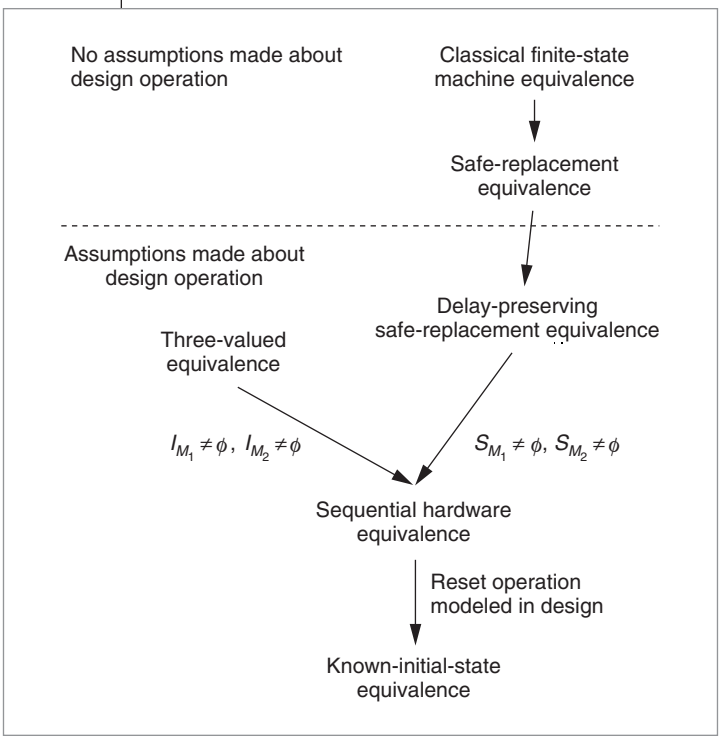


Figure 8. Relationship of sequential-equivalence notions for machines M_1 and M_2. Labeling on the arrows indicate conditions under which the relationship holds.

Consequently, a state in one machine need not be equivalent to a state in another. What is guaranteed by safe replaceability is that any start-up state in M_2 behaves like some start-up state of M_1 and vice versa.

In Figure 7, M_3 is not safe-replacement equivalent to M_1 because in state v_4, input sequence 11 results in output sequence 11. This I/O behavior cannot be reproduced from any state in M_1. On the other hand, machine M_4 is safe-replacement equivalent to M_1.

Three-valued safe equivalence

Three-valued safe equivalence is a relaxed form of safe-replacement equivalence that allows some freedom in the machine's transient behavior.[9] Two machines M_1 and M_2 are three-valued equivalent if for any input sequence $\pi \in \Sigma^k$, $k > 0$, their outputs belong to the set $\{(0, 0), (1, 1), (X, X)\}$ when π is applied from an unknown state (all flip-flops initialized to X). The intuition behind this definition is the following: If during initialization the original machine produces a 0 or 1 depending on its power-up state, the replacement machine can produce any value for that output. However, if during initialization the output is predictable (that is, always 0 or always 1 from all power-up states), the replacement machine must preserve that behavior. After initialization, the original machine will be in a known state, and consequently its output will be predictable. Thus, any replacement machine must have identical behavior after initialization.

Other notions of sequential equivalence are defined in the literature. We refer the reader to Singhal's dissertation for definitions of delay and delay-preserving equivalence.[13] Figure 8 illustrates the relationship of the various equivalence notions. An arrow from A to B indicates that if M_1 and M_2 are equivalent under A, they are equivalent under B.

SEQUENTIAL SYNTHESIS and optimization of digital circuits demand robust equivalence-checking algorithms. The heuristic approaches reviewed here increase the capacity of the basic algorithms so that they can handle circuits with hundreds to thousands of memory elements. Although this is a significant enhancement, it is still inadequate for state-of-the-art designs. More research in this area is essential to successfully deploying sequential-equivalence-checking tools in industrial environments. ■

■ References

1. C.E. Leiserson and J.B. Saxe, "Retiming Synchronous Circuitry," *Algorithmica*, vol. 6, no. 1, 1991, pp. 5-35.
2. Z. Kohavi, *Switching and Finite Automata Theory*, McGraw-Hill, 1978.
3. J. Hartmanis and R.E. Stearns, *Algebraic Structure Theory of Sequential Machines*, Prentice-Hall, 1966.
4. O. Coudert, B. Berthet, and J. Madre, "Verification of Synchronous Sequential Machines Based on Symbolic Execution," *Proc. Int'l Workshop Automatic Verification Methods for Finite State Systems*, LNCS 407, Springer, 1989, pp. 365-373.
5. F. Somenzi, "Binary Decision Diagrams," *Calculational System Design*, vol. 173, NATO Science Series F, IOS Press, 1999, pp. 303-366.
6. R. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," *IEEE Trans. Computers*, vol. 35, no. 8, Aug. 1986, pp. 677-691.
7. M. Sheeran, S. Singh, and G. Stalmarck, "Checking Safety Properties Using Induction and a SAT Solver," *Proc. 3rd Int'l Conf. Formal Methods in Computer-Aided Design* (FMCAD 2000), LNCS 1954, Springer, 2000, pp. 108-125.
8. P. Bjesse and K. Claessen, "SAT-Based Verification without State Space Traversal," *Proc. 3rd Int'l Conf. Formal Methods in Computer-Aided Design* (FMCAD 2000),

LNCS 1954, Springer, 2000, pp. 372-389.
9. S.-Y. Huang, K.-T. Cheng, and K.-C. Chen, "Verifying Sequential Equivalence Using ATPG Techniques," *ACM Trans. Design Automation of Electronic Systems*, vol. 6, no. 2, Apr. 2001, pp. 244-275.
10. D. Stoffel and W. Kunz, *Structural FSM Traversal—Theory and a Practical Algorithm*, tech. report 005/1997, Dept. of Computer Science, Univ. of Potsdam, Germany, 1997.
11. C.A.J. van Eijk, "Sequential Equivalence Checking Based on Structural Similarities," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. 19, no. 7, July 2000, pp. 814-819.
12. C. Pixley, "A Theory and Implementation of Sequential Hardware Equivalence," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. 11, no. 12, Dec. 1992, pp. 1469-1494.
13. V. Singhal, *Design Replacements for Sequential Circuits*, doctoral dissertation, Univ. of California, Berkeley, Electronics Research Laboratory, College of Engineering, 1996.

Maher N. Mneimneh is pursuing a PhD in computer science and engineering at the University of Michigan. His research interests include formal verification of VLSI systems: model checking, combinational and sequential equivalence checking, binary decision diagrams, and satisfiability solvers. Mneimneh has a BE in computer and communications engineering from the American University of Beirut and an MS in computer science and engineering from the University of Michigan.

Karem A. Sakallah is a professor of electrical engineering and computer science at the University of Michigan, Ann Arbor. His research interests include CAD with emphasis on simulation, timing verification and optimal clocking, logic and layout synthesis, Boolean satisfiability, and design verification. Sakallah has a BE in electrical engineering from the American University of Beirut and an MSEE and a PhD in electrical and computer engineering from Carnegie Mellon University. He is a fellow of the IEEE and a member of the ACM and Sigma Xi.

■ Direct questions and comments about this article to Maher N. Mneimneh, University of Michigan, Dept. of Computer Science and Engineering, 1301 Beal Avenue, Ann Arbor, MI 48109-2122; maherm@umich.edu.