SAT-based Sequential Depth Computation

Maher Mneimneh, Karem Sakallah
University of Michigan
{maherm,karem}@umcih.edu

Abstract – **Determining the depth of sequential circuits is a crucial step towards the completeness of bounded model checking proofs in hardware verification. In this paper, we formulate sequential depth computation as a logical inference problem for Quantified Boolean Formulas. We introduce a novel technique to simplify the complexity of the constructed formulas by applying simple transformations to the circuit netlist. We also study the structure of the resulting simplified QBFs and construct an efficient SAT-based algorithm to check their satisfiability. We report promising experimental results on some of the ISCAS 89 benchmarks.**

I. Introduction

The crucial demands to design bug-free products caused today's typical verification teams to be the size of entire design teams ten years ago. Still, faulty chips are fabricated, forcing first customers to find out about undetected glitches, and causing severe company losses for replacement. The infamous Pentium floating-point bug is one example.

By far, the most common verification paradigm in use today is simulation. Such a verification strategy is becoming increasingly untenable, however, because of time-to-market pressures and high costs of design errors discovered following product release. In addition, simulation-based verification cannot guarantee that a design is bug-free; at best, simulation can reduce the probability of releasing a buggy design to acceptable levels but can never completely certify design correctness.

Formal hardware verification is a promising alternative to traditional simulation. Formal hardware verification can be broadly classified into two major categories: property checking and equivalence checking. Property checking verifies the correctness of a property pertaining to a hardware model, whereas equivalence checking verifies the equivalence between two models. Most of the techniques utilized for property checking apply as well to equivalence checking.

There is a rich literature on the different approaches to property checking of hardware systems. These approaches can be broadly divided into two groups: theorem proving and model checking.

Model checking [5] is a predominant technique in formal hardware verification. In model checking, the system to be verified is modeled as a Kripke structure, and the specification is modeled as a formula in some temporal logic. Binary Decision Diagrams (BDDs) [3] are used to symbolically represent sets of states and transition relations in Kripke structures.

Despite the enormous success of symbolic model checking techniques [4], most practical designs were still outside the scope of existing tools. BDD-based model checking techniques heavily rely on the availability of efficient representation and manipulation using BDDs. However, BDDs can grow exponentially in size. Exponential BDD sizes are caused by either "state explosion" or humongous transition relations. If such a situation occur, BDD-based approaches are doomed to fail.

To surmount this, several solutions were suggested of which Bounded Model Checking (BMC) [1, 2] is one. BMC checks safety and liveness properties of a system using a satisfiability (SAT) solver. In BMC, a Propositional formula is generated that is satisfiable if there exists a path ending in a state that satisfies the property checked. If such a state can not be found at depth i, the search is repeated at depth $i + 1$. One advantage of BMC over BDD-based model checking is the absence of a state keeping-device. In BDD-based model checking, at each iteration a BDD keeps track of states. This might cause BDD explosion. On the other hand, BMC requires a reasoning engine with the capability of handling a larger number of variables. In fact, each unfolding of the transition relation corresponds to adding a new set of variables. In general, this is not a major obstacle for SAT-solvers. One major drawback of BMC, however, is its lack of completeness. In other words, we do not know when to stop incrementing the value of i.

In this paper, we tackle the issue of completeness for BMC. We take a fresh look at the problem of determining the sequential depth of a circuit using search-based techniques. Search-based techniques eliminate the need for a BDD-based state-keeping device and consequently avoid the possibility of exponential space requirements. Once the sequential depth of a circuit is determined, it can guide BMC or any similar approach when checking safety properties (e.g., invariants.) As a result, BMC can now be used not only to find bugs, but also to prove correctness.

The contributions in this paper are two-fold. First, we formulate sequential depth computation an a logical inference problem for Quantified Boolean Formulas (QBF) and present a powerful reduction technique that simplifies the formulas by introducing safe modifications to the circuit's state transition graph. The modifications are safe in the sense that the sequential depth is left intact but the complexity of computing it is dramatically reduced. Second, we study the structure of the resulting simplified QBFs and construct a SAT-based approach to solve their satisfiability. For these formulas, the algorithm is far more efficient than state-of-the-art QBF solvers.

The paper is organized as follows. In section II, we review relevant work on sequential depth computation and show how it relates to the particular approach we describe. Section III introduces finite state machines, state transition graphs, and the notion of sequential depth using graph-theoretic concepts. Section IV formulates sequential depth computation as a logical inference problem for QBF. In section V, we describe how to simplify the resulting QBFs by introducing state transition graph modifications that keep the sequential depth intact. Section VI is devoted to describing a SAT-based algorithm that efficiently solves the simplified QBFs. Experimental results on some of the ISCAS 89 benchmarks are discussed in section VII and the paper is concluded in section VIII with some pointers to future work.

II. Previous Work

We review some of the literature related to sequential depth computation using search-based approaches. Although BDD-based approaches that perform breadth-first search of the state transition graph compute sequential depth, we will not describe such techniques here for the sake of brevity.

The problem of determining when it is safe to stop going into deeper states when checking safety properties has been addressed by Sheeran et. al in [9]. The authors observe that no more iterations should be performed if no new states exist in future iterations. They check for that by searching for a path of length i starting at the initial state (a path, as defined in the next section, should have distinct states.) If no path exists, then no new states exist at depth i and the iteration is stopped. On the other hand, if a path exists, then a new state exists and the iteration continues to larger values of i. The authors use a SAT solver to check the existence of a path at a given depth. A major disadvantage of that approach is the over-approximations of the sequential depth that might result. The existence of a path from the initial state to some state does not guarantee that this state has not been encountered on a different and shorter path. The authors provide a complete formulation in terms of shortest paths but no experimental evaluation is presented.

In [10], random simulation is utilized to provide an estimate of the sequential depth. A queue is used to keep track of states. The algorithm starts by storing the initial state in the queue and setting its depth to 0. At each iteration, a state is dequeued, and a random input is used to compute its next state. This step is repeated for a number of times determined by a threshold. If the resulting next state has not been encountered before, the state is queued and its depth is set to that of the present state plus one. When the threshold is reached, a new state is dequeued and the above procedure is repeated. When the queue is empty, the algorithm terminates. The sequential depth is the depth of the last stated dequeued. The above algorithm samples the state space and consequently might result in under as well as over-approximations of the sequential depth. An improvement of the algorithm that profiles the toggle activity of state variables is also provided.

In [8], Plaisted et al. present an algorithm for solving the satisfiability problem for QBFs and discuss its use in determining fixpoints of repetitive systems. However, no experimental analysis showing the effectiveness of their QBF solver on such systems is presented.

III. Preliminaries

A finite-state machine (FSM) M is defined as a 6-tuple $M = (Q, \Sigma, \Delta, \delta, \lambda, Q^0)$ where Q is a finite set of *states*, Σ is the *input alphabet*, Δ is the *output alphabet*, $\delta: Q \times \Sigma \rightarrow Q$ is the state-transition function, $\lambda: Q \times \Sigma \rightarrow \Delta$ is the output function, and Q^0 is the set of initial states.

Synchronous Sequential circuits are modeled using FSMs. A synchronous sequential 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 combinational part of the circuit is made up of k internal signals ($w_1, w_2, \ldots, w_k$) representing the outputs of combinational gates. A clock signal clk synchronizes the operation of the memory elements. Each of these signals takes one of two possible values 0 or 1. We will refer to $x_1, x_2, \ldots, x_m$ as the *input variables*, $z_1, z_2, \ldots, z_l$ as the *output variables*, $y_1, y_2, \ldots, y_n$ as the *state variables*, and $w_1, w_2, \ldots, w_k$ as the *internal variables*.

The state-transition function $\delta: Q \times \Sigma \rightarrow Q$ determines the next state of the machine based on its current state and inputs. The output function $\lambda: Q \times \Sigma \rightarrow \Delta$ determines the machine's output based on its current state and inputs. We can write:

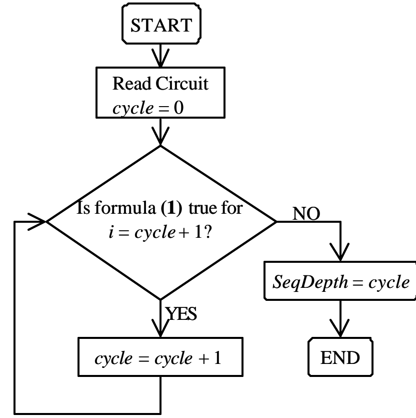

Figure 1: Flowchart of the algorithm for computing sequential depth

$$y^+ = \delta(y, x) \qquad z = \lambda(y, x)$$

where $x \equiv (x_1, \ldots, x_m)$, $y \equiv (y_1, \ldots, y_n)$, $y^+ \equiv (y_1^+, \ldots, y_n^+)$, $z \equiv (z_1, \ldots, z_l)$, $\delta \equiv (\delta_1, \ldots, \delta_n)$, and $\lambda \equiv (\lambda_1, \ldots, \lambda_l)$.

We define the transition relation of the circuit as:

$$T(y^+, y, x) = \bigwedge_{i=1}^{n} (y_i^+ \odot \delta_i(y, x))$$

where $\odot$ is the XNOR function. Intuitively, $T(y^+, y, x) = 1$ if there is a transition from state y to state y^+ on input x. Otherwise, $T(y^+, y, x) = 0$. If the internal variables $w_1, w_2, \ldots, w_k$ are part of the transition relation, we write:

$$T(y^+, y, x, w) = \bigwedge_{i=1}^{n} (y_i^+ \odot \delta_i(y, x, w))$$

The state transition graph of an FSM M, STG(M), is a labeled directed graph $\langle V, E \rangle$ where each vertex $v \in V$ corresponds to a state s_i of M (and is labeled with s_i), and each edge $e \in E$ between two vertices 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.

Given a directed graph $G\langle V, E \rangle$. A *walk* of length k is a succession of k directed edges $v_0v_1, v_1v_2, \ldots, v_{k-1}v_k$. If the directed edges of a walk are different, the walk is called a *trail*. A trail whose vertices are all different is called a *path*.

Given a directed graph $G\langle V, E \rangle$. The distance between two vertices $v, u \in V$, denoted $d(v, u)$ is the *shortest* path between v and u. The *eccentricity* of v, $ecc(v)$ is the longest of all the shortest paths between v and every other vertex in V. We can write $ecc(v) = max_u d(v, u)$ for all $u \in V$. The radius of G is the minimum eccentricity among all its vertices, $radius(G) = min_v(ecc(v))$ for all $v \in V$. The *diameter* of G is the maximum eccentricity among all its vertices, $diameter(G) = max_v(ecc(v))$.

Given a directed graph $G\langle V, E \rangle$. Consider two vertices v_0 and v_k. v_k is *reachable* from v_0 if a path exists from v_0 to v_k. In addition, each vertex is reachable from itself.

Consider an FSM M with a single initial state s_0. We define the *sequential depth* of M as the eccentricity of its initial state s_0 in the corresponding state transition graph STG(M), and denote it by $SeqDepth(M)$. Consider two states s_0 and s_i. If there is a path from

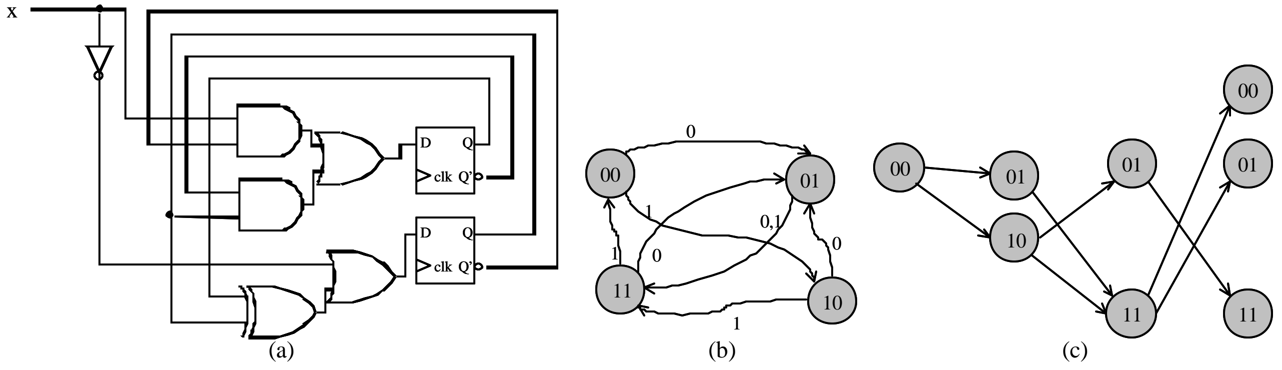

Figure 2: (a) A sequential circuit (b) its corresponding STG and (c) computation tree.

s_0 to s_i we call s_0 the *start-state* and s_i the *end-state* for that path. For all paths we consider, the start-state is the initial state unless a different start-state is specified. Similarly, if there is a walk between s_0 and s_i, s_0 is the start-state and s_i is the end-state for that walk.

IV. Sequential Depth Computation By Logical Inference

Since the definition of depth depends on paths in the state-transition graph, we will define paths in terms of the transition relation of the circuit. Let $T(\boldsymbol{y}^+, \boldsymbol{y}, \boldsymbol{x}, \boldsymbol{w})$ be the transition relation where $\boldsymbol{x} = (x_1, x_2, \ldots, x_m)$ are the input variables, $\boldsymbol{y} = (y_1, y_2, \ldots, y_n)$ current-state variables, $\boldsymbol{y}^+ = (y_1^+, y_2^+, \ldots, y_n^+)$ are next-state variables, and $\boldsymbol{w} = (w_1, w_2, \ldots, w_k)$ are internal variables. We define $walk_i(\boldsymbol{y}^0, \boldsymbol{y})$ as follows:

$$walk_1(\boldsymbol{y}^0, \boldsymbol{y}) = \exists \boldsymbol{x}\boldsymbol{w} \;\; T(\boldsymbol{y}, \boldsymbol{y}^0, \boldsymbol{x}, \boldsymbol{w})$$

and for $i > 1$,

$$walk_i(\boldsymbol{y}^0, \boldsymbol{y}) = \exists \boldsymbol{y}^1\boldsymbol{y}^2\ldots\boldsymbol{y}^{i-1}$$

$$walk_1(\boldsymbol{y}^0, \boldsymbol{y}^1) \cdot walk_1(\boldsymbol{y}^1, \boldsymbol{y}^2) \cdot \ldots \cdot walk_1(\boldsymbol{y}^{i-1}, \boldsymbol{y})$$

Similarly, we define $path_i(\boldsymbol{y}^0, \boldsymbol{y})$ as follows:

$$path_1(\boldsymbol{y}^0, \boldsymbol{y}) = \exists \boldsymbol{x}\boldsymbol{w} \;\; T(\boldsymbol{y}, \boldsymbol{y}^0, \boldsymbol{x}, \boldsymbol{w}) \cdot (\boldsymbol{y} \neq \boldsymbol{y}^0)$$

$$path_i(\boldsymbol{y}^0, \boldsymbol{y}) = \exists \boldsymbol{y}^1\boldsymbol{y}^2\ldots\boldsymbol{y}^{i-1}$$

$$path_1(\boldsymbol{y}^0, \boldsymbol{y}^1) \cdot path_1(\boldsymbol{y}^1, \boldsymbol{y}^2) \cdot \ldots \cdot path_1(\boldsymbol{y}^{i-1}, \boldsymbol{y}) \cdot$$

$$(\boldsymbol{y}^2 \neq \boldsymbol{y}^0)$$

$$\ldots$$

$$(\boldsymbol{y} \neq \boldsymbol{y}^{i-2}) \cdot (\boldsymbol{y} \neq \boldsymbol{y}^{i-3}) \cdot \ldots \cdot (\boldsymbol{y} \neq \boldsymbol{y}^0)$$

where for $\boldsymbol{x} = (x_1, x_2 \ldots, x_n)$, and $\boldsymbol{y} = (y_1, y_2, \ldots, y_n)$

$$\boldsymbol{x} \neq \boldsymbol{y} \leftrightarrow (x_1 \oplus y_1) + (x_2 \oplus y_2) + \ldots + (x_n \oplus y_n)$$

Intuitively, $path_i(\boldsymbol{y}^0, \boldsymbol{y}) = 1$ if $\boldsymbol{y}$ is reachable from $\boldsymbol{y}^0$ in a path of length i and is 0 otherwise.

We have previously defined sequential depth as the eccentricity of the initial state, or in other words, as the longest of all shortest paths between the initial state and every other reachable state. Thus, one way to compute sequential depth is to find a reachable state that has the largest distance from the initial state (recall that distance between two vertices is the shortest path between them.) Such a procedure can be performed iteratively. We start at depth 1 and proceed as follows. At depth i, we look for a state whose distance from the initial state is i. Such a state will have the property that no path of length less than i has that state as an end-state; otherwise, the path of length i is not the shortest for that state. A logical formulation of that property is

$$\exists \boldsymbol{y} \; I(\boldsymbol{y}^0) \cdot path_i(\boldsymbol{y}^0, \boldsymbol{y}) \cdot \neg path_{i-1}(\boldsymbol{y}^0, \boldsymbol{y}) \cdot \ldots \cdot \neg path_1(\boldsymbol{y}^0, \boldsymbol{y}) \quad (1)$$

where $I(\boldsymbol{y}^0)$ is the predicate of the initial state.

If the above formula is true, the algorithm proceeds to deeper iterations since new states might be found whose shortest distance from the initial state is greater than i. The algorithm terminates when no state satisfying the above property can be found. In other words, the algorithm terminates when the above logical formula is false. A flowchart for computing sequential depth based on this algorithm is illustrated in Figure1.

As an example, consider the sequential circuit and its corresponding STG in Figure2(a) and (b). Figure2(c) shows the computation tree that results from unrolling the STG. We start by setting $cycle = 0$. The above formula searches for a path of length 1 whose end-state is different from the initial state. $00 \rightarrow 01$ is one such path. As a result, $cycle$ is incremented to 1. Next, the formula searches for a path length 2 whose end-state is not also an end-state for any path of length 1. $00 \rightarrow 01 \rightarrow 11$ is such a path and $cycle$ is incremented to 2. In the third iteration, the formula searches for a path of length 3 with an end-state that is not an end-state for any path of length 2 or 1. The formula is false in that case; no such state can be found as is apparent from the computation tree of Figure2(c). The algorithm terminates returning a sequential depth of 2.

Searching for a state s that satisfies the above formula is complicated by the fact that at a given depth i, the search should consider paths of length j for all values of j less than i to ensure that the path to s is the shortest. This can be very expensive rendering the algorithm inapplicable but for simple circuits. In the next section we introduce a major simplification that significantly reduces the complexity of the formula to be checked.

V. Simplified Sequential Depth Computation

Checking the satisfiability of (1) suffers from two major complications. First, the quantifiers and negation in $path_i(\boldsymbol{y}^0, \boldsymbol{y})$ transform the above formula into a QBF that can not be simply checked using a SAT solver. Second, when searching for a new state at depth i, we have to check all depths less than i to ensure that the state is on a shorter path. This check can result in very expensive computations.

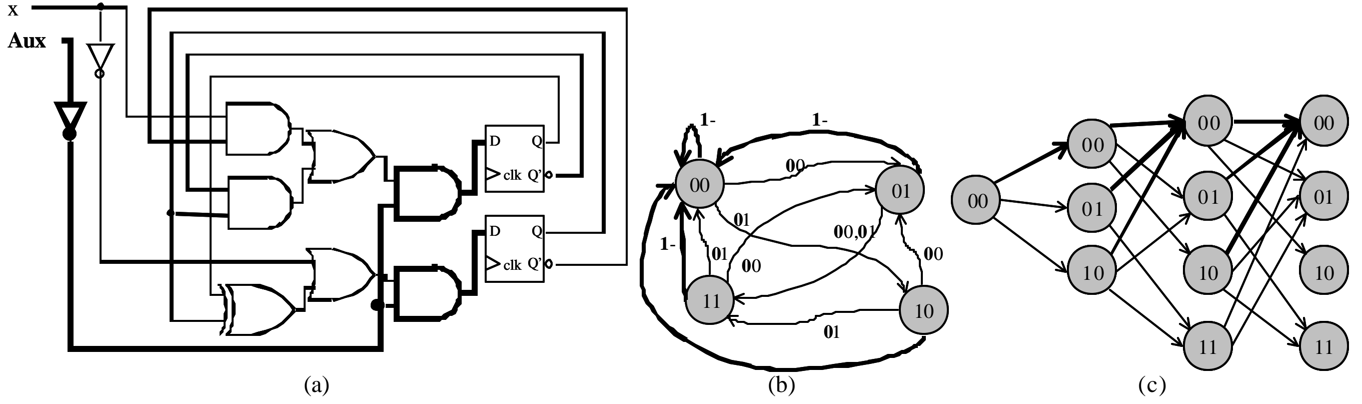

Figure 3: (a) The modified sequential circuit (b) its corresponding STG and (c) computation tree.

In this section we address the second problem. We would like to simplify the logical formula so that we do not have to go all the way back to depths of length 1 when searching for a new state. To better understand the problem, consider again the computation tree in Figure2(c). Assume we are checking for new states at depth 3. Possible paths of length 3 are $00 \rightarrow 10 \rightarrow 01 \rightarrow 11$ and $00 \rightarrow 10 \rightarrow 11 \rightarrow 01$. Note that $00 \rightarrow 10 \rightarrow 11 \rightarrow 00$, $00 \rightarrow 01 \rightarrow 11 \rightarrow 00$, and $00 \rightarrow 01 \rightarrow 11 \rightarrow 01$ are not paths since each has a repeated state. Consider the path $00 \rightarrow 10 \rightarrow 01 \rightarrow 11$. This is not the shortest path to state 11 since the path $00 \rightarrow 01 \rightarrow 11$ of length 2 ends in that state. For 11 it was enough to check paths of length 2 to decide that it is not on a shortest path during iteration 3. Now, consider the path $00 \rightarrow 10 \rightarrow 11 \rightarrow 01$. Although there is no path of length 2 having 01 as an end-state, $00 \rightarrow 01$ is such a path having length 1. In that case, to conclude that $00 \rightarrow 10 \rightarrow 11 \rightarrow 01$ is not the shortest path to 01 we had to go back to paths of length 1.

We can avoid the situation encountered in the last example by introducing a simple modification to the STG being traversed. Consider the STG shown in Figure3(b) derived from that of Figure2(b) by introducing new edges from every state to the initial state 00. The additional edges are shown in bold. Transitions along these edges take place whenever a new auxiliary input has the value 1. When the auxiliary input is 0, the operation of the original machine is not altered. The computation tree corresponding to the new STG is shown in Figure3(c). Consider again the path $00 \rightarrow 10 \rightarrow 11 \rightarrow 01$. We can check walks of length 2 only to deduce that 01 is not a new state. The walk $00 \rightarrow 00 \rightarrow 01$ establishes such proof. In the STG of Figure3(b), we do not have to go back and search all depths smaller than the one considered; at depth i, it is enough to check for walks of length $i-1$ to prove or disprove the existence of a shortest path of length i. This results in substantial reductions when checking our formula

Next, we present two theorems that prove the correctness of the above approach.

Theorem 1 Given an FSM M having an initial state s_0, and its corresponding STG, $STG(M)$. Let M' be a new FSM whose STG is derived from that of M by adding transitions from every state to the initial state s_0. Then $SeqDepth(M) = SeqDepth(M')$.

The above theorem ensures that by adding transitions to the initial state, the sequential depth of the machine is not altered.

Theorem 2 Given an FSM M having an initial state s_0. Let STG(M) be such that there is a transition from every state in M to s_0. Let s_i be a state such that there is a path from s_0 to s_j of length l, $l > 0$. Then there is a walk from s_0 to s_j of length m for every m such that $m \geq l$.

The above theorem ensures that every state reachable at depth l through a path from the initial state of an FSM M, will be also reachable at any depth $m \geq l$ through a walk from the initial state.

For brevity reasons, we omit the proofs of the above theorems.

The above theorems enable us to reduce the formula for checking the sequential depth to:

$$\exists \boldsymbol{y} \quad I(\boldsymbol{y}^0) \cdot path_n(\boldsymbol{y}^0, \boldsymbol{y}) \cdot \neg walk_{n-1}(\boldsymbol{y}^0, \boldsymbol{y}) \qquad (2)$$

because by Theorem 2:

$$(path_{n-1}(\boldsymbol{y}^0, \boldsymbol{y}) + \ldots + path_1(\boldsymbol{y}^0, \boldsymbol{y})) \rightarrow walk_{n-1}(\boldsymbol{y}^0, \boldsymbol{y})$$

In addition, from basic graph theory we have:

$$walk_{n-1}(\boldsymbol{y}^0, \boldsymbol{y}) \rightarrow (path_{n-1}(\boldsymbol{y}^0, \boldsymbol{y}) + \ldots + path_1(\boldsymbol{y}^0, \boldsymbol{y}))$$

Thus,

$$(path_{n-1}(\boldsymbol{y}^0, \boldsymbol{y}) + \ldots + path_1(\boldsymbol{y}^0, \boldsymbol{y})) \leftrightarrow walk_{n-1}(\boldsymbol{y}^0, \boldsymbol{y})$$

and consequently,

$$(\neg path_{n-1}(\boldsymbol{y}^0, \boldsymbol{y}) \cdot \ldots \cdot \neg path_1(\boldsymbol{y}^0, \boldsymbol{y})) \leftrightarrow \neg walk_{n-1}(\boldsymbol{y}^0, \boldsymbol{y})$$

To obtain the STG with transitions from every state to the initial state, we have to modify the given circuit. The new circuit corresponding to that of Figure2(a) is shown in Figure3(a). The bold parts corresponds to the additional logic. An auxiliary input is added. Each next-state variable of the original circuit is ANDed with the negated auxiliary input and the output is fed to the latches. Whenever the auxiliary input is 1, the next state will be the initial state (in that case 00). Whenever the auxiliary input is 0, the machine's operation is identical to the original circuit. Note that if the initial state is different from $00\ldots0$, the logic should be modified accordingly.

Note also that these additional transitions might be present in the original circuit if the initialization is modeled at the logic level. In that case, the above modifications do not need to be added.

VI. SAT-based Solution to Depth Computation

In this section, we present an efficient satisfiability checking procedure for formulas of structure similar to (2).

Recall that the above formula is in QBF form and consequently can not be checked immediately using a SAT solver. One possibility is to use a QBF solver. However, even formulas resulting from small circuits are out of the scope of state-of-the-art QBF solvers.
Let's take a closer look at the above formula. $path_n(\boldsymbol{y}^0, \boldsymbol{y})$ represents n iterations of the transition relation $T(\boldsymbol{y}^+, \boldsymbol{y}, \boldsymbol{x}, \boldsymbol{w})$ with additional constraints that the states along the paths are distinct. $walk_{n-1}(\boldsymbol{y}^0, \boldsymbol{y})$ represents $n-1$ iterations of the transition relation. $path_n(\boldsymbol{y}^0, \boldsymbol{y})$ and $walk_{n-1}(\boldsymbol{y}^0, \boldsymbol{y})$ do not share any intermediate variables. The only variables shared between the two formulas are those corresponding to the initial state $\boldsymbol{y}^0$ and the final state $\boldsymbol{y}$. This suggests an efficient sequential depth computation algorithm that is presented in the flowchart of Figure4.
The algorithms starts by reading the sequential circuit and setting the variable $cycle = 0$. Let's consider iteration i of the algorithm. At iteration i, $cycle = i$, and the algorithm has to check the satisfiability of (2). Our algorithm decomposes checking the satisfiability of (2) into two simpler checks. The algorithm starts by constructing a CNF formula representing a path of length $i+1$ starting at the initial state. A SAT solver is used to find a satisfiable assignment for the constructed formula which corresponds to:

$$I(\boldsymbol{y}^0) \cdot path_n(\boldsymbol{y}^0, \boldsymbol{y}) \qquad (3)$$

If (3) is unsatisfiable, we automatically know that (2) is unsatisfiable. Thus, the algorithms terminates and $SeqDepth = i$.
If on the other hand, (3) is satisfiable, we construct a CNF formula representing a walk of length i and set its end-state to $\boldsymbol{y}^s$, the end-state on the path of length $i+1$ that was returned as a satisfying solution for (3). The formula corresponds to:

$$I(\boldsymbol{y}^0) \cdot walk_{n-1}(\boldsymbol{y}^0, \boldsymbol{y}^s) \qquad (4)$$

A SAT solver is used to check (4). If (4) is unsatisfiable, then there is no walk of length i starting at the initial state and ending in $\boldsymbol{y}^s$. Consequently, $\boldsymbol{y}^s$ is a new state at depth $i+1$ that satisfies (2). Thus, $cycle$ is incremented and the whole procedure is repeated again.
If (4) is satisfiable, then $\boldsymbol{y}^s$ do not satisfy (2) and we have to search for another solution. We repeat the procedure again by try to find a satisfiable assignment to (3) other than $\boldsymbol{y}^s$. Consequently, we add a constraint to (3) enforcing the SAT solver to find a state different from $\boldsymbol{y}^s$. Whenever no state can satisfy (3), we terminate setting the sequential depth to i. Note that all the added constraints to (3) at depth i are used at all the following iterations. This ensures that no time is wasted looking for states that are not on shortest paths.
Let's consider the execution of the algorithm on the circuit of Figure3(a), whose computation tree is shown in Figure3(c).
The algorithm starts by setting $cycle = 0$. A CNF formula corresponding to $path_1(00, \boldsymbol{y})$ is constructed. No learned clauses are present at this time to be added. We find a satisfying assignment for $path_1(00, \boldsymbol{y})$. One such solution is $00 \to 01$. $\boldsymbol{y} = 01$ is a new state so we set $cycle = 1$.
Next, we construct a formula corresponding to $path_2(00, \boldsymbol{y})$. A satisfying solution to $path_2(00, \boldsymbol{y})$ is $00 \to 01 \to 11$ where $\boldsymbol{y} = 11$. To check whether 11 is a new state, we construct a formula corresponding to $walk_1(00, 11)$ Such a formula is unsatisfiable (no walk of length 1 exists that terminate in 11.) Thus 11 is a new state; we set $cycle = 2$.
Again, we construct a formula corresponding to $path_3(00, \boldsymbol{y})$. A satisfying solution is $00 \to 10 \to 01 \to 11$ where $\boldsymbol{y} = 11$. We check whether $walk_2(00, 11)$ is satisfiable. Sine there is a walk

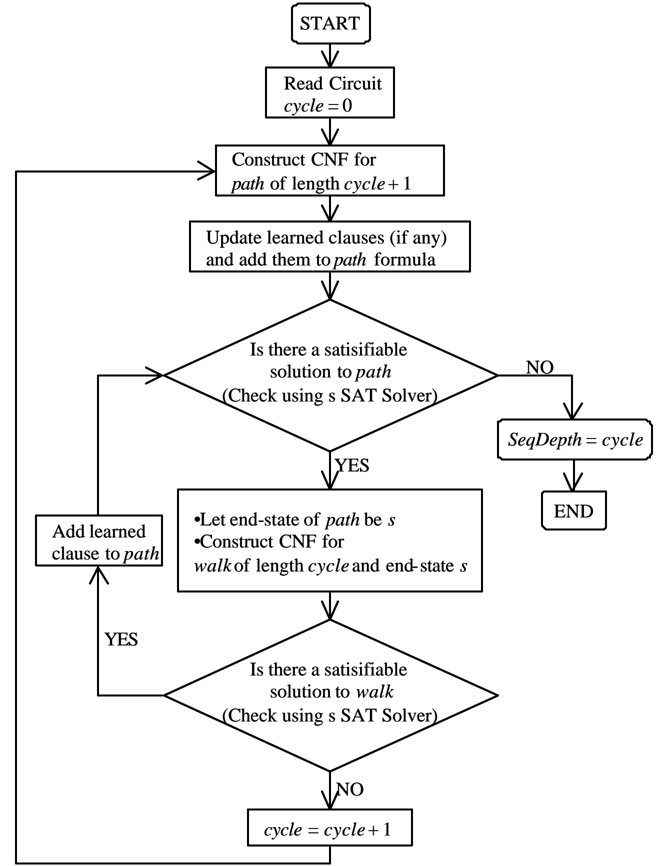

Figure 4: Flowchart of the enhanced algorithm for determining sequential depth

$00 \to 01 \to 11$, $walk_2(00, 11)$ is satisfiable and 11 is not a new state. We add a constraint to $path_3(00, \boldsymbol{y})$ forcing $\boldsymbol{y}$ to be different The resulting formula is $path_3(00, \boldsymbol{y}) \cdot (\boldsymbol{y} \neq 11)$. Satisfiability check is repeated on the new formula. A new satisfying solution is $00 \to 10 \to 11 \to 01$ where $\boldsymbol{y} = 01$. We check whether there is a walk of length 2 ending in 01. The check returns $00 \to 00 \to 01$. Thus 01 is not a new state. Again, we add the constraint $\boldsymbol{y} \neq 01$ to $path_3(00, \boldsymbol{y}) \cdot (\boldsymbol{y} \neq 11)$. Checking the new formula $path_3(00, \boldsymbol{y}) \cdot (\boldsymbol{y} \neq 11) \cdot (\boldsymbol{y} \neq 01)$ yields no satisfying assignment. Thus no new states exist at depth 3. The algorithm terminates; the sequential depth is 2.

VII. Experimental Results

To experimentally evaluate the effectiveness of our algorithm, we implemented it in C++ and used Chaff [7] as the underlying SAT solver. We report our results on the ISCAS 89 benchmarks. All experiments were conducted on a 2 GHz Pentium 4 machine having 1 GB of RAM and running the Linux operating system. To our knowledge, there is no published data on the sequential depth of ISCAS benchmarks computed using SAT-based search only and to which we can compare our results. For that reason, we compare our results to those obtained in [6] where a combination of SAT, BDDs, and partitioning techniques are used to perform reachability analysis. For this comparison, we note the following. First, the techniques in [6] compute the whole set of reachable states using breadth-first search (BFS). In that case, sequential depth is the number of BFS iterations. Our algorithm does not compute the set of reachable states. Second, in [6], the authors set the

time limit to 100,000 seconds whereas we set the time limit to 5,000 seconds.

TABLE I:Experimental data on a sample of the ISCAS 89 benchmarks

Circuit	Sequential Depth	Time (sec)	Maximum Depth in [6]
s298	18	19.3	-
s386	7	0.18	-
s499	21	1.07	-
s510	46	144.81	-
s641	6	97.03	-
s713	6	126.94	-
s820	10	2.51	-
s953	10	102.23	-
s1196	2	232.84	-
s1269	7	5000	10(c)
s1423	26	5000	15
s1488	21	96.87	-
s3271	14	5000	17(c)
s3330	7	5000	9(c)
s5378	19	5000	45(c)
s6669	5	5000	3

Experimental results for a sample of the ISACS benchmarks are reported in TableI. The name of the circuit appears in column 1. Columns 2 and 3 show the sequential depth and the running time for our algorithm. In case the time limit was reached, we report the maximum depth attained. Column 4 reports the maximum number of reachability steps in [6]; a (c) next to a number indicates that reachability was carried to completion. Empty cells in column 4 indicate non-reported data in [6].

For the small benchmarks, up to s1196, our algorithm is very effective in determining the sequential depth. The least time reported is 0.18 seconds for s386, and the longest time is 232.84 seconds for s1196. Although s386 is deeper than s1196, s386 has 13 reachable states while s1196 has 2616. Consequently, more searching is needed in the case of s1196 before reaching a fixpoint at the sequential depth.

For s1269, s3271, s3330, and s5378, the time limit of 5000 seconds was reached. Although reachability analysis completed successfully for these benchmarks in [6], the maximum depth we attained in 5000 seconds is very close to the sequential depth (except for s5379.) This maximum depth can be very effective in guiding BMC. In fact, the BMC tool should keep iterating for at least the value of the maximum depth.

The results for s1423 and s6669 are interesting. Reachability analysis in [6] could not complete: a maximum depth of 15 and 3 were obtained for s1423 and s6669. However, we were able to reach a depth of 26 for s1423 and 5 for s6669. When combined with BMC, we are able to check for properties at depths that BDDs can not handle.

VIII. Conclusion

Determining the depth of sequential circuits enables search-based verification techniques such as bounded model check to prove correctness of designs rather than just find bugs.

To achieve completeness for Bounded Model Checking, we presented a formulation of sequential depth computation as a logical inference problem for Quantified Boolean Formulas. We studied the structure of the resulting formulas and showed that their complexity can be drastically reduced by modifying the netlist of the circuit they are constructed from. We proved that such modifications leave the sequential depth intact and consequently are safe to apply. We also presented an efficient SAT-based algorithm to solve QBFs resulting from our simplifications. Our experimental results showed that SAT-based depth computation can sometimes be a good alternative to BDD-based techniques.

Since our algorithm learns states that violate some property and adds them to the clause database one at a time, its memory requirements are exponential in the worst case (this situation arises when the number of states is exponential.) To surmount this, we are currently investigating the possibility of adding sets of states at a time. In addition, we are exploring two other directions to improve our approach. We are studying the possibility of applying abstractions that preserve the sequential depth of a circuit. These abstractions can help simplify the complexity of the QBFs to be checked. We are also investigating better algorithms to solve the formulas that arise during depth computation. These algorithms try to utilize symmetries in the transition relation and along different paths to simplify the satisfiability check.

References

[1] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu, "Symbolic Model Checking Using SAT Procedures instead of BDDs," in 36th Design Automation Conference, 1999.

[2] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolic Model Checking without BDDs," in TACAS'99, 1999.

[3] R. E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," in *IEEE Transactions on Computers*, 35(8), pp. 677-691, August 1986.

[4] J. R. Burch, E. M. Clarke, D. E. Long, K. L. McMillan, and D. L. Dill, "Symbolic Model Checking for Sequential Circuit Verification," in *IEEE Transactions on Computer-Aided Design of Integrated Circuits* 13(4), pp. 401-424, April 1994.

[5] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic Verification of Finite-state Concurrent Systems Using Temporal Logic Specifications," in *ACM Transactions on Programming Languages and Systems*, 8(2), pp. 244-263, 1986.

[6] A. Gupta, Z. Yang, P. Ashar, L. Zhang, and S. Malik, "Partition-Based Decision Heuristics for Image Computation Using SAT and BDDs," in *Proceedings of the International Conference on Computer-Aided Design*, 2001.

[7] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an Efficient SAT solver," in *Proceedings of the Design Automation Conference*, 2001.

[8] D. A. Plaisted, A. Biere, and Y. Zhu, "A Satisfiability Procedure for Quantified Boolean Formulae," submitted for publication, Discrete Applied Mathematics.

[9] M. Sheeran, S. Singh, and G. Stalmarck, "Checking Safety Properties Using Induction and a SAT-solver," in *Formal Methods in Computer Aided Design*, 2000.

[10] C.-C. Yen, K.-C Chen, and J.-Y. Jou, "A Practical Approach to Cycle Bound Estimation for Property Checking," in *11th IEEE/ACM International Workshop on Logic Synthesis*, pp. 149-154, June 2002.