AQUILA: An Equivalence Verifier for Large Sequential Circuits

Shi-Yu Huang Kwang-Ting Cheng
Department of Electrical & Computer Engineering
University of California, Santa Barbara
Santa Barbara, CA 93106
huang@yellowstone.ece.ucsb.edu timcheng@ece.ucsb.edu

Kuang-Chien Chen
Fujitsu Labs. of America
3350 Scott Blvd. Bldg 34
Santa Clara, CA 95054
kchen@fla.fujitsu.com

Abstract

In this paper, we address the problem of verifying the equivalence of two sequential circuits. A hybrid approach that combines the advantages of BDD-based and ATPG-based approaches is introduced. Furthermore, we incorporate a technique called partial justification to explore the sequential similarity between the two circuits under verification to speed up the verification process. Compared with existing approaches, our method is much less vulnerable to the memory explosion problem, and therefore can handle larger designs. The experimental results show that in a few minutes of CPU time, our tool can verify the sequential equivalence of an intensively optimized benchmark circuit with hundreds of flip-flops against its original version.

1. Introduction

State-of-the-art synthesis tools optimize circuits with respect to various constraints such as area, performance, power dissipation and testability. These tools can apply sequential transformations to a circuit and therefore may result in an optimized network with a different number of flip-flops [14,4,20]. Even though these transformations are correct by construction, the software programs that implement these transformations are highly complicated and may not be error-free. Therefore, verifying the correctness of an optimized circuit is necessary. For those circuits that have been manually changed to satisfy the timing or power dissipation requirements in the late design cycle, verification is even more important. Recently, several efficient approaches for verifying a sequential circuit based on Binary Decision Diagrams (BDD's) have been proposed [6,5,21,8]. In these approaches, the circuits are regarded as finite state machines and characterized by a transition relation and a set of output functions using BDD's. A product machine is constructed and its state space is traversed. Most of them assume a reset state, and employ a breadth-first traversal algorithm to compute the set of reachable states. The equivalence of these two machines can be proved by checking the tautology of every primary output of the product machine. Due to the memory explosion problem, these approaches can easily fail for large designs. The use of BDD dynamic ordering techniques [19] can extend the capability of these state-traversal-based approaches. However, this extension still has limitations in handling large designs.

Exploring the structural similarity between the two circuits under verification has been shown to be effective to reduce the complexity of verifying combinational circuits [2,13,12,18,11,15]. We extended this idea to verify sequential circuits [9]. In this extension, sequential Automatic Test Pattern Generation (ATPG) techniques [3,7] are used to identify equivalent flip-flop pairs and equivalent internal signal pairs. A computational model called *miter* [7,2] is constructed and sequential backward justification technique is employed. We implicitly regard the verification problem as a search process for an input sequence that can differentiate the given two circuits, instead of trying to compute all reachable states in one shot. The problem is divided into a set of easier sub-problems: verifying equivalent flip-flop pairs and internal signal pairs. But because the search space for each sub-problem is still very high, we further developed several techniques to cut down the search space. A pre-processing algorithm was recently developed for retimed circuits [10] to increase the internal structural similarity of the retimed and the reference circuits. Through such an algorithm, the complexity of verifying retimed circuits is dramatically reduced.

The efficiency of this approach relies on the techniques for identifying the equivalent internal signal pairs. Based on an observation that a high percentage of equivalent pairs can be identified by only considering a small subcircuit surrounding the candidate signal pair, we developed an algorithm for identifying these signal pairs for combinational circuits by only constructing local BDD's. The BDD's are incrementally expanded on demand during the verification process. This enhancement makes our approach less sensitive to the circuit's structural similarity.

Two other new developments further strengthen our approach. First, a more effective procedure that combines the advantages of the BDD-based and ATPG-based techniques is introduced. Secondly, we extend the local BDD-based verification engine for sequential circuits. This technique, referred to as *partial justification*, enables our framework to handle much larger sequential designs.

The rest of this paper is organized as follows. In Section 2, we review our earlier framework. In Section 3, we describe our new procedure and the algorithm of partial justification. We present the experimental results in Section 4 and give the conclusion in Section 5.

2. Preliminary

2.1 Computational model

A sequential circuit is regarded as a set of interconnected components with primary inputs and primary outputs. These components could be flip-flops or logic gates. If the circuit cannot be reset to an unique reset state externally, several different definitions of sequential equivalence, such as *post-synchronization equivalence* [16], *safe replaceability* [17], and *3-valued safe replaceability* [9] can be used. We assume the circuit has an external reset state in this paper, but the discussion can be easily extended for checking other definitions of equivalence. The verification is performed on the computational model called *miter* as shown in Fig. 1. Each primary output pair is connected to an exclusive-OR gate (whose output is denoted as g). The specification and implementation are denoted as C_1 and C_2 respectively. For simplicity without losing generality, we assume C_1 and C_2 are both single output circuits, and their outputs

ASP-DAC '97
0-89791-851-7$5.00

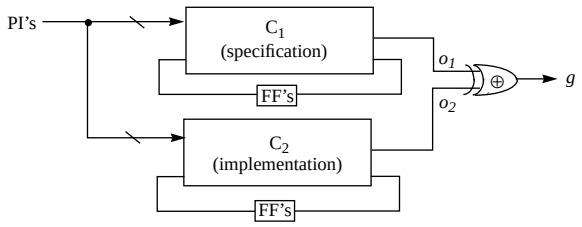


Fig. 1: The computational model for equivalence checking.

are o_1 and o_2 respectively. The problem is to decide if the response of o_1 is equivalent to o_2 for all possible input sequences.

Definition 1 (Signal pair): (a_1, a_2) is called a signal pair if a_1 is a signal of C_1, and a_2 is a signal of C_2.

2.2 The basic ATPG-based framework

We review the procedure of using a sequential ATPG program for verification in this sub-section. We first build the miter of the two circuits, and then perform a modified ATPG process to search for a test of g stuck-at-0 fault using the reverse-time processing technique on the iterative array model as shown in Fig. 2. Since we

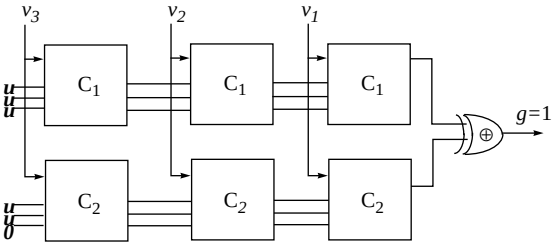


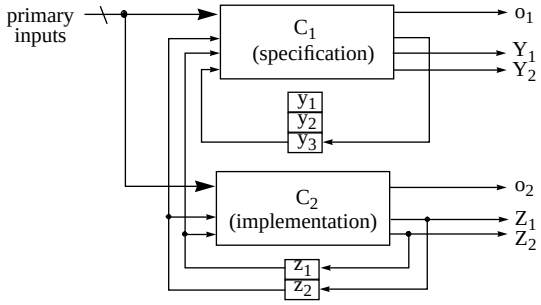
Fig. 2: A example of backward justification.

are dealing with an output fault, the forward fault-effect propagation is not necessary. The search process consists of two stages: (1) fault injection, and (2) sequential backward justification.

During the process of backward justification, value assignments at the present state lines of each time-frame is called a state requirement, denoted as *sr*, which can be considered as a state-cube. We decompose a state requirement into two parts: the state requirement for C_1, sr_1; and the state requirement for C_2, sr_2. For example, $sr = (sr_1 \mid sr_2) = (uuu \mid uu0)$, where u means "no requirement" at that particular state bit. We monitor each state requirement generated during this process. If the reset state, e.g., (000 | 000), is contained in a newly generated state requirement, then it indicates a test sequence has been found and the justification process terminates. In the example of Fig. 2, $T = (v_3v_2v_1)$ is a test sequence for g stuck-at-0 because the reset state is contained in a state requirement ($uuu \mid uu0$) after the justification of 3 time frames. On the other hand, if the state requirement does not contain the reset state, it needs to be further justified by expanding another time frame until it is eventually justified or proven unjustifiable. If all the state requirements ever generated in this backward justification process are proven unjustifiable, then no distinguishing sequence exists and C_1 is equivalent to C_2. Note that this is a recursive search process, and the number of state requirements may grow rapidly. For the cases when the given two circuits are indeed equivalent, the above branch-and-bound search would be very time-consuming, and thus, requires some speed-up techniques. We developed a procedure to take advantage of the similarity between the circuits for this purpose. This procedure has two major steps: (1) identify equivalent flip-flop pairs, and (2) identify equivalent internal pairs.

2.3 Identifying equivalent flip-flop pairs

The main idea of identifying equivalent flip-flop pairs is to pair up candidate equivalent pairs by name comparison, if possible, or by simulation of some random patterns at the beginning. A signal pair will be a candidate pair if their responses to the random sequence are identical. After the initial candidate set has been constructed, we employ an iterative process to incrementally filter out those *false* candidate flip-flop pairs (flip-flop pairs that are actually inequivalent). At each iteration of this filtering process, we construct a new intermediate model as shown in Fig 3. In this



candidate flip-flop pairs: $\{(y_1, z_1), (y_2, z_2)\}$
candidate next-state line pairs: $\{(Y_1, Z_1), (Y_2, Z_2)\}$
* y_1 and y_2 are replaced by z_1 and z_2 respectively in this model.

Fig. 3: The intermediate model for identifying equivalent FF pairs.

model, every candidate flip-flop pair is *assumed equivalent* and connected together. In Fig. 3, the initial candidate set is $\{(y_1, z_1), \{y_2, z_2)\}$. Also the inputs of the candidate flip-flop pairs are treated as pseudo primary outputs and referred to as *candidate next-state line pairs* (NS-pairs), e.g., $\{(Y_1, Z_1), \{Y_2, Z_2)\}$ in Fig. 3. Equivalence checking is performed for every candidate NS-pair on this model. If a candidate NS-pair is not equivalent in this model, then its associated flip-flop pair is a false candidate, and should be removed from the candidate set. As soon as a false candidate flip-flop pair is identified, the assumption of the present state line pairs (PS-pairs) of all candidate pairs including the identified false pair being equivalent is proven incorrect. Therefore, the identified false pair is removed and a new iteration starts with a smaller candidate set. This iterative process will stop when all candidate NS-pairs are proven equivalent in the intermediate model. It can be proved by induction that when such a stable condition is reached, all candidate pairs that survive this filtering process are indeed equivalent, and those flip-flop pairs that are filtered out are indeed inequivalent.

This filtering procedure may need several iterations to reach a stable condition. However, it is still much more efficient than a primitive method that identifies equivalent flip-flop pairs directly without assuming the equivalence of every candidate PS-pair. The efficiency of this method is due to three main reasons: (1) The *search space* of the distinguishing sequences for each candidate NS-pair is dramatically reduced because of the bounding effects created by merging the PS lines of every candidate pair. (2) The

number of flip-flops are reduced in the intermediate model at each iteration. (3) Merging the candidate flip-flop pairs creates more common supports for the two circuits' combinational portions. Therefore, a lot of *sequentially* equivalent internal signal pairs are converted into *combinationally* equivalent. These pairs can then be explored much more easily.

At each iteration of the above procedure, checking the equivalence of each candidate NS-pair is still computationally expensive for large circuits. Identifying and merging the internal equivalent signal pairs can further improve the efficiency. In [11], a process of verifying combinational equivalence is performed in stages. The internal equivalent pairs are identified from the primary input side towards the primary output side using a modified ATPG. Once an equivalent pair is identified, they are merged to prune the *miter* and speed up the subsequent verification process. We integrate the BDD-based and the ATPG-based techniques to extend this idea for sequential circuits. We identify not only combinationally equivalent but also sequentially equivalent signal pairs.

2.4 Identifying equivalent internal signal pairs

The ATPG-based approach is efficient if the given two circuits have significant structural similarity. But for circuits optimized through extensive transformations, a pure ATPG-based technique may not be sufficient. For such cases, proving equivalent internal signal pairs can be done by constructing *local BDD's* [11,15]. Fig. 4

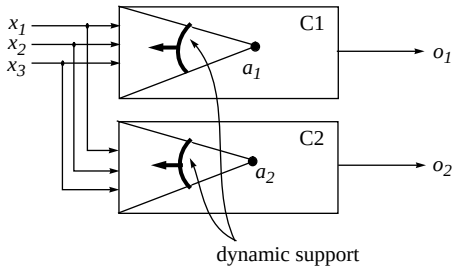


Fig. 4: The dynamic support that expands towards the primary inputs for verifying the equivalence of (a_1, a_2).

shows the idea of proving an internal equivalent pair for combinational circuits by constructing local BDD's. Suppose the primary inputs are x_1, x_2, and x_3; and the target pair is (a_1, a_2). We select a set of supporting signals in the fanin cones of signals a_1 and a_2. If we can prove that for all value combinations at these supporting signals, (a_1, a_2) is an equivalent pair, then they are indeed equivalent. Otherwise, we expand the support towards the inputs incrementally until a proper support is found to prove the equivalence, or the number of expansions exceeds a pre-defined limit. Our experiments on the combinational benchmark circuits optimized by extensive scripts (such as *script.rugged*) show that a large number of equivalent pairs can be identified using this sufficient criterion. In Section 3, we will show how to incorporate this idea into the basic ATPG-based framework for sequential circuits.

3. Hybrid Algorithm

In this Section, we describe the integrated procedure for verifying sequential circuits. The procedure is based on the combination of a robust local BDD-based engine and the effective search for a counter-sequence using ATPG. The overall flow is shown in Fig. 5. There are three major phases: (1) Run simulation with a random sequence to find the candidate equivalent flip-flop pairs and internal pairs. (2) Identify equivalent flip-flop pairs and simplify the miter. This phase is very efficient and is based on both BDD-based and ATPG-based techniques. In detail, this phase is an iterative process and has three steps for each iteration. First, we assume that every candidate flip-flop pair is equivalent, and therefore connect the present state lines (PS lines) of each candidate pair together. Second, based on this assumption, internal equivalent pairs are identified and merged in stages. Third, the next state lines of each candidate flip-flop pairs are verified to decide if the process has stabilized. If yes, exit the loop. Otherwise, remove the false candidate pair from the candidate list and continue with another iteration. Note that this is a monotone filtering process, and thus, the process will always terminate. Identifying equivalent signal pairs in this process is based on a symbolic backward justification technique to be described in the next subsection. (3) Check the equivalence of each primary output pair. We use the local BDD-based approach followed by an ATPG search if necessary.

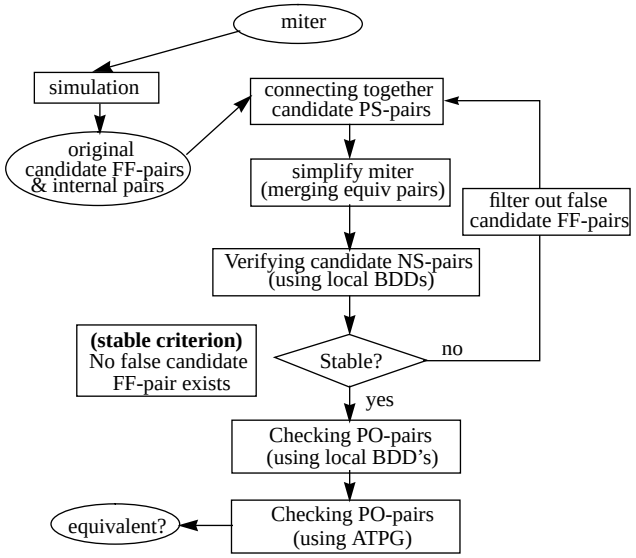


Fig. 5: Overall algorithm.

3.1 Symbolic backward justification

In this sub-section, we describe a symbolic algorithm to check the equivalence of a signal pair (a_1, a_2). The process is performed on the iterative array model of the miter. The difference is that we do the backward justification in a symbolic way instead of performing a branch-and-bound search.

At the last time frame of this procedure, we treat a_1 and a_2 as pseudo outputs and compute the set of value combinations at primary inputs (PI's) and present state lines (PS's) that can differentiate this target pair (i.e., setting g to '1'). Fig. 6 shows an illustration. We call the characteristic function of this set as the discrepancy function at time frame 0, denoted as $Disc^0(a_1, a_2)$. We smooth out all the primary inputs to obtain a new function in terms of the present state lines only. This new function, denoted as $SR^0(a_1, a_2)$, characterizes the set of the state requirements at the last time

frame for differentiating (a_1, a_2). For the rest of the paper, the superscript of a notation indicates the index of the associated time frame. The index of the last time frame is 0 and it increase in a backward manner as shown in Fig. 6.

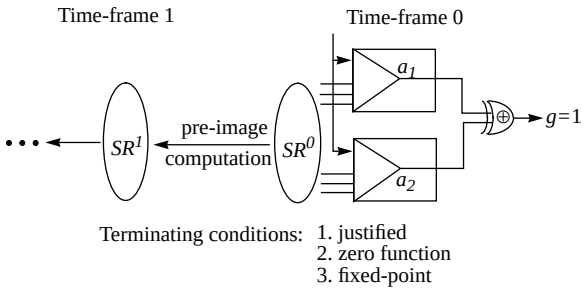


Fig. 6: Symbolic backward justification for checking if $a_1 = a_2$.

Once we have derived $SR^0(a_1, a_2)$, the symbolic backward justification process in the following should be performed to decide whether $SR^0(a_1, a_2)$ can be justified. This is done by a series of pre-image computations. At each time frame i, (i >0), $SR^i(a_1, a_2)$ is derived by computing the pre-image of $SR^{(i-1)}(a_1, a_2)$ followed by smoothing out all primary inputs. A number of existing techniques using BDD's for pre-image computation can be directly applied. Similar to our original ATPG-based framework, the terminating conditions should be checked at each time frame to see if another backward time frame expansion is needed. Assume the current time frame index is i, then the three terminating conditions are as follows. (1) Justified condition: when the reset state is contained in $SR^i(a_1, a_2)$. (2) Conflict condition: when $SR^i(a_1, a_2)$ is the zero function. (3) Fixed-point (or cyclic) condition: when $SR^i(a_1, a_2)$ is contained in $SR^j(a_1, a_2)$ for some $j < i$. During this symbolic backward justification, if conditions (2) or (3) are encountered, then the target signal pair are equivalent. Otherwise they are not equivalent. The algorithm is detailed in Fig. 7.

```
Symbolic-Equivalence-Checking (a1, a2)
{
    compute discrepancy function Disc^0(a1, a2) at the last time frame.
    derive the state requirement function SR^0(a1, a2) by smoothing out PI's.
    SR = SR^0(a1, a2);
    /*------ symbolic backward justification ------*/
    while(1)
      c = check_terminating_conditions(SR);
      switch(c){
        case JUSTIFIED: return(DIFFERENT);
        case CONFLICT: return(EQUIVALENT);
        case FIXED-POINT: return(EQUIVALENT);
        default: New=compute_preimage_and_smooth_out_PI's(SR); break;
      }
      SR = SR ∪ New;
    }
}
```

Fig. 7: The symbolic algorithm to check the equivalence of a signal pair.

3.2 Partial justification

The above symbolic backward justification may cause memory explosion for circuits with a large number of flip-flops. Hence, we further incorporate a simple technique called *partial justification* to target for some practical designs. This technique can be used during the process of identifying the equivalent flip-flops pairs, internal equivalent pairs, and the process of checking primary output equivalences.

Assume that all the candidate present state line pairs are connected together at each time frame and the identified equivalent internal pairs in the fanins of (a_1, a_2) have been merged. Now the verification process proceeds to check if a_1 is equivalent to a_2. At the last time frame, instead of computing discrepancy function in terms of the PI's and PS's directly, we select a local cutset in this air's fanins. This cutset is denoted as λ_0. We use $Disc_{\lambda_0}^{0}(a_1, a_2)$ to denote the characteristic function of the set of value combinations at λ_0 that can differentiate (a_1, a_2). Note that all cutset signals n λ_0 should be previously identified as equivalent to their corresponding signals, and have been merged with their corresponding signals. Suppose we incrementally backward expand this cutset for a number of logic levels within the same time frame and still cannot prove the target pair is equivalent (i.e., $Disc_{\lambda_0}^{0}(a_1, a_2)$ is not the zero function), then we stop and pessimistically assume that they are not combinationally equivalent. We then proceed to the partial justification process to check if they are sequentially equivalent.

To start this process, we first smooth out the signals in λ_0 that are not present state lines from $Disc_{\lambda_0}^{0}(a_1, a_2)$ to obtain a partial state requirement function $SR_{\lambda_0}^{0}(a_1, a_2)$. This function characterizes a necessary condition that should be satisfied at the present state lines to differentiate (a_1, a_2). Similar to the complete symbolic justification, a number of time frames may need to be explored until one of the three terminating conditions is met. For simplicity, we only select one cutset at each time frame i, denoted as λ_i. We derive $SR_{\lambda_i}^{i}(a_1, a_2)$ from $SR_{\lambda_{i-1}}^{i-1}(a_1, a_2)$ by pre-image computation and smoothing out non-PS supporting signals in λ_i. After this process of partial justification, if the objective of differentiating (a_1, a_2) turns out to be unjustifiable, then (a_1, a_2) is equivalent. But on the contrary, if the partial state requirement function can be justified, then no conclusion can be made.

Selecting a good cutset is essential for improving the performance of verification. Similar to the combinational cases (as shown in Fig. 4), we expand the cutset dynamically from the target pairs towards the primary inputs and present state lines at each time frame. We use a simple heuristic that looks backwards for a number of levels to select a cutset with a small number of signals. Our experiments showed that this simple heuristic for expanding the cutset dynamically results in a very high percentage of equivalent pairs even for a fully optimized circuit. When no conclusion can be reached using this heuristic, the false negative problem may occur, i.e., our algorithm fails to identify an equivalent pair even though they are indeed equivalent. In that case, if the target pair are internal signals, we assume they are inequivalent pessimistically. If the target pair is a flip-flop pair or primary output pair, then we rely on the ATPG to resolve the false negative problem or to find a distinguishing sequence.

3.3 Example

Fig. 8 shows an example. This example has four primary inputs x_1, x_2, x_3, and x_4. C_1 and C_2 contain sequential sub-networks, sub_1 and sub_2, respectively. Suppose the outputs of these two sub-networks, s_1 and s_2, are proven equivalent. In addition to these two sub-networks, C_1 (C_2) has two flip-flops denoted as Z_1 and Z_2 (Y_1 and Y_2) respectively. We make no differentiation between a logic gate and its output signal except for the flip-flops. The outputs of Z_1 and Z_2 (Y_1, and Y_2) are the present state lines and denoted as z_1 and z_2 (y_1 and y_2). The verification procedure is detailed as follows.

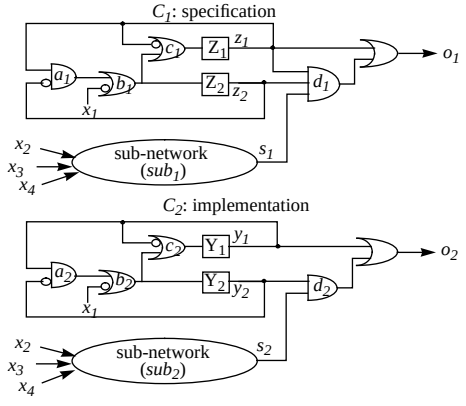


Fig. 8: An example to illustrate the entire procedure.

(Step 1): Perform simulation for a large number of random vectors to find the set of candidate flip-flop pairs $\{(Z_1, Y_1), (Z_2, Y_2)\}$, and the set of candidate internal pairs $\{(a_1, a_2), (b_1, b_2), (c_1, c_2), (d_1, d_2), (s_1, s_2)\}$.

(Step 2): Identify the equivalent flip-flop pairs.

(2.1) Assume the candidate present state lines pairs $\{(z_1, y_1), (z_2, y_2)\}$ are equivalent and connect each PS-pair together.

(2.2) Verify the equivalence of $\{(Z_1, Y_1), (Z_2, Y_2)\}$. Signal pairs $\{(a_1, a_2), (b_1, b_2), (c_1, c_2)\}$ can be easily verified as equivalent incrementally, and so can $\{(Z_1, Y_1), (Z_2, Y_2)\}$. Hence, we conclude that the assumption made in (2.1) is correct and the process has stabilized. (Z_1, Y_1) and (Z_2, Y_2) are indeed equivalent flip-flop pairs.

(Step 3): Check the equivalence of primary output pair (o_1, o_2).

(3.1) Further explore the similarity by checking if (s_1, s_2) and (d_1, d_2) are equivalent pairs. Suppose that (s_1, s_2) is equivalent and, thus, we replace s_1 by s_2. The process moves on to check (d_1, d_2). Fig. 9 shows a snapshot of the miter at this moment. Suppose we select $\lambda = \{y_1, y_2, s_2\}$ as the cutset. Then the distinguishing vector at this cutset is $\{(y_1, y_2, s_2) = (0, 1, 1)\}$. Let the characteristic function of this set be $Disc_\lambda^0(d_1, d_2)$. Since signal s_2 is not a present state line, it should be smoothed out from $Disc_\lambda^0(d_1, d_2)$ to derive the set of state requirement, which would be $\{(y_1, y_2) = (0, 1)\}$. None of the three terminating conditions is met. Hence the backward justification process starts. At the next time frame, the pre-image of $\{(Y_1, Y_2) \mid (0, 1)\}$ is an empty set, and thus a conflict has been reached. Signal pair (d_1, d_2) is sequentially equivalent and can be merged together.

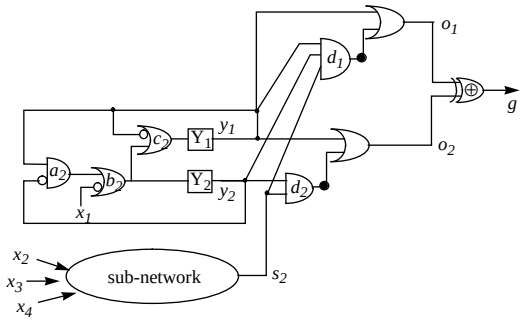


Fig. 9: A snapshot of miter before checking equivalence of (d_1, d_2).

(3.2): Check output pair (o_1, o_2) using local BDD's. They can be proved equivalent by selecting $\{y_1, d_2\}$ as the cutset.

(Step 4): Run an ATPG program on the simplified miter. This is not necessary in this example.

4. Experimental Results

We have implemented the described framework in C language in the SIS environment [8]. It also integrates a sequential ATPG program *stg3* [1]. We tested it on a suite of ISCAS89 benchmark circuits optimized by *script.rugged* script using SIS. The optimization process reduces the number of flip-flops for circuits s641, s713, s5378, s13207, and s15850. For these circuits, combinational verification programs cannot be used even if the inputs (outputs) of the flip-flops are treated as pseudo-outputs (pseudo-inputs). Our program automatically verified that these optimized circuits are indeed sequentially equivalent to their original version directly.

Table 1 shows the results of the experiments. The meaning of each column is explained as follows. (1) *# nodes (original / opti-*

Table 1: Results of verifying circuits optimized by *script.rugged* of SIS

Circuit	# nodes original / optimized	# FFs orig. / opt.	# internal equiv. pairs (comb / seq)	# equivalent FF-pairs (comb. / seq.)	# equivalent PO-pairs (comb. / seq.)	Verify Time (sec)
s208	66 / 42	8 / 8	9 / 0	8 (8 / 0)	1 (1 / 0)	1
s298	75 / 65	14 / 14	17 / 0	14(14/ 0)	6 (6 / 0)	2
s344	114 / 102	15 / 15	29 / 0	15 (15 / 0)	11(11 / 0)	1
s349	117 / 101	15 / 15	29 / 0	15 (15 / 0)	11 (11 / 0)	1
s382	101 / 92	21 / 21	28 / 0	21 (21 / 0)	6 (6 / 0)	1
s386	118 / 60	6 / 6	7 / 0	6 (6 / 0)	7 (7 / 0)	1
s400	108 / 88	21 / 21	27 / 0	21 (21 / 0)	6 (6 / 0)	1
s420	140 / 84	16 / 16	19 / 0	16 (16 / 0)	1 (1 / 0)	1
s444	121 / 85	21 / 21	25 / 0	21 (21 / 0)	6 (6 / 0)	1
s510	179 / 115	6 / 6	8 / 0	6 (6 / 0)	7 (7 / 0)	1
s526	141 / 106	21 / 21	31 / 0	21 (21 / 0)	6 (6 / 0)	1
*s641	128 / 100	**19 / 17**	4 / 0	17 (2 / 15)	23 (16 / 7)	11
*s713	154 / 99	**19 / 17**	4 / 0	17 (2 / 15)	23 (14 / 9)	12
s820	256 / 137	5 / 5	16 / 0	5 (5 / 0)	19 (19 / 0)	1
s832	262 / 130	5 / 5	16 / 0	5 (5 / 0)	19 (19 / 0)	1
s838	288 / 162	32 / 32	39 / 0	32 (32 / 0)	1 (1 / 0)	1
s1196	389 / 273	18 / 18	40 / 0	17 (17 / 0)	14 (14 / 0)	3
s1238	429 / 286	18 / 18	46 / 0	17 (17 / 0)	14 (13 / 1)	4
s1423	491 / 390	74 / 74	109 / 0	72 (72 / 0)	5 (5 / 0)	3
s1488	550 / 309	6 / 6	29 / 0	6 (6 / 0)	19 (19 / 0)	3
s1494	558 / 305	6 / 6	30 / 0	6 (6 / 0)	19 (19 / 0)	3
*s5378	1074 / 858	**164 / 162**	223 / 0	162 (142 / 20)	49 (49 / 0)	12
s9234	1081 / 599	135 / 135	156 / 0	135 (135 / 0)	39 (39 / 0)	5
*s13207	2480 / 1175	**490 / 453**	301 / 32	419 (379 / 40)	121 (77 / 44)	122
*s15850	3379 / 2435	**563 / 540**	608 / 20	526 (523 / 3)	87 (73 / 14)	634
s35932	12492 / 7050	1728 / 1728	2048 / 0	1728 (1728 / 0)	320 (320 / 0)	75
s38417	8623 / 7964	1464 / 1464	1860 / 0	1464 (1464 / 0)	106 (106 / 0)	241

* Circuits whose numbers of flip-flops are reduced after the logic optimization process.

mized): the numbers of nodes of the original and optimized circuits after being cleaned up by SIS command *"sweep"* and then decomposed into AND/OR gates. (2) *# FFs (orig. / opt.)*: the numbers of flip-flops in the original and optimized circuits. For instance, s13207 has 490 flip-flops, but only 453 left in the optimized circuit. (3) *# equiv. internal pairs (comb / seq)*: the number of internal signal pairs that are identified as combinationally equivalent and sequentially equivalent in our program respectively. Identifying these pairs play an important role in reducing the run-time complexity. (4) *# equivalent FF-pairs (comb. / seq.)*: the numbers of equivalent flip-flop pairs that are verified as combinationally equivalent and sequentially equivalent respectively. Among the (490 / 453) flip-flops of original and optimized s13207, 419 pairs are identified as equivalent using our program, where 379 pairs are combinationally equivalent and 40 pairs are sequentially equivalent. (5) *# equivalent PO-pairs (comb. / seq.)*: the numbers of combinationally and sequentially equivalent primary output pairs.

The verification time on a Sun-sparc5 with 128-Mbyte memory in seconds is given in the last column. Table 2 shows the results of using our program to verify the circuits after sequential redundancy removal. Among the total 23 circuits, thirteen (including s1423, s5378 and s9234) are sequentially equivalent instead of combinationally equivalent to their original version. It is worth mentioning that, to our knowledge, no pure FSM-traversal technique has successfully verified the ISCAS89 benchmark circuits larger than s1423.

Table 2: Results of verifying circuits after redundancy removal.

Circuit	# nodes original / optimized	# FFs orig. / opt.	# internal eq. pairs (comb / seq.)	# equivalent FF-pairs (comb. / seq.)	# equivalent PO-pairs (comb. / seq.)	Verify Time (sec)
s208	66 / 66	8 / 8	65 / 0	8 (8 / 0)	1 (1 / 0)	1
*s298	75 / 74	14 / 14	63 / 0	14(11 / 3)	6 (6 / 0)	1
*s344	114 / 113	15 / 15	102 / 0	15 (15 / 0)	11(10 / 1)	1
*s349	117 / 113	15 / 15	102 / 0	15 (15 / 0)	11 (11 / 1)	1
s382	101 / 100	21 / 21	56 / 0	21 (9 / 12)	6 (6 / 0)	2
*s386	118 / 111	6 / 6	78 / 0	6 (4 / 2)	7 (2 / 5)	1
*s400	108 / 101	21 / 21	55 / 0	21 (9 / 12)	6 (6 / 0)	2
s420	140 / 140	16 / 16	139 / 0	16 (16 / 0)	1 (1 / 0)	1
*s444	121 / 105	21 / 21	49 / 0	21 (9 / 11)	6 (6 / 0)	7
s510	179 / 179	6 / 6	172 / 0	6 (6 / 0)	7 (7 / 0)	1
*s526	141 / 139	21 / 21	90 / 0	20 (11 / 9)	6 (6 / 0)	20
*s641	128 / 120	**19 / 18**	97 / 0	**17** (2 / 15)	23 (16 / 7)	2
s713	154 / 111	19 / 19	79 / 0	19 (19 / 0)	23 (23 / 0)	2
s820	256 / 256	5 / 5	237 / 0	5 (5 / 0)	19 (19 / 0)	1
s832	262 / 256	5 / 5	217 / 0	5 (5 / 0)	19 (19 / 0)	1
s838	288 / 288	32 / 32	287 / 0	32 (32 / 0)	1 (1 / 0)	1
*s1196	389 / 387	**18 / 18**	364 / 0	**17** (17 / 0)	14 (12 / 2)	2
*s1238	429 / 386	**18 / 18**	305 / 0	**17** (17 / 0)	14 (12 / 2)	8
s1488	550 / 550	6 / 6	550 / 0	6 (6 / 0)	19 (19 / 0)	3
s1494	558 / 548	6 / 6	523 / 0	6 (6 / 0)	19 (19 / 0)	2
*s1423	491 / 463	**74 / 74**	441 / 0	**71** (68 / 3)	5 (5 / 0)	96
*s5378	1074 / 919	**164 / 139**	891 / 18	**139**(139 / 0)	49 (43 / 6)	120
*s9234	1081 / 1067	**135 / 131**	1052 / 0	**125** (122 / 3)	39 (38 / 1)	46

* Redundancy removed circuits that are sequentially equivalent, but not combinationally equivalent, to their original version.

5. Conclusion

Existing state-traversal-based verification approaches for equivalence checking are subject to combinatorial explosion, and thus, only applicable to small to medium-sized circuits. In an attempt to handle larger circuits, we propose a hybrid method that combines the advantages of local BDD-based and ATPG-based approaches. To speed up the verification process, we devise a robust engine to explore the sequential similarity between circuits under verification based on the idea of partial justification. Because of using local BDD's, our approach is less sensitive to the degree of structural similarity as compared to the pure ATPG-based approaches. We presented the experimental results of verifying large ISCAS89 benchmark circuits that have been fully optimized by SIS, and circuits after sequential redundancy removal to show the capability of this promising approach.

References

[1] M. Abramovici, M. A. Breuer, and A. D. Friedman, "Digital Systems Testing and Testable Design," *IEEE Press* (1990).

[2] D. Brand, "Verification of Large Synthesized Designs," *Proc. Int'l Conf. on CAD*, pp. 534-537 (Nov. 1993).

[3] W.-T. Cheng, "The BACK Algorithm for Sequential Test Generation," *Proc. Int'l Conference on Computer Design* (ICCD-88), pp. 66-69 (Oct. 1988).

[4] K.-T. Cheng, "Redundancy Removal for Sequential Circuits Without Reset States," *IEEE Trans. on CAD*, pp. 652-667 (Jan. 1993).

[5] H. Cho, G. D. Hachtel, S. W. Jeong, B. Plessier, E. Schwarz, and F. Somenzi, "ATPG Aspects of FSM Verification," *Proc. Int'l Conf. on CAD*, pp. 134-137 (Nov. 1990).

[6] O. Coudert, C. Berthet, and J. C. Madre, "Verification of Synchronous Sequential Machines Based on Symbolic Execution," *Automatic Verification Methods for Finite State System, LNCS no. 407*, Springer Verlag (1990).

[7] A. Ghosh, S. Devadas, and A. R. Newton, "Test Generation and Verification for Highly Sequential Circuits," *IEEE trans. on CAD*, pp. 652-667 (May 1991).

[8] Y. V. Hoskote, "Formal Techniques for Verification of Synchronous Sequential Circuits", Dept. of ECE, Univ. of Texas at Austin, Ph.D. Dissertation (Dec. 1995).

[9] S.-Y. Huang, K.-T. Cheng, K.-C. Chen, and U. Glaeser, "An ATPG-Based Framework for Verifying Sequential Equivalence," *Proc. Int'l Test Conference*, pp. 865-874 (Oct. 1996).

[10] S.-Y. Huang, K.-T. Cheng, and K.-C. Chen, "On Verifying the Correctness of a Retimed Circuit," *Proc. Great-Lake Symposium on VLSI*, pp. 277-280 (March 1996).

[11] S.-Y. Huang, K.-C. Chen, and K.-T. Cheng, "Error Correction Based on Verification Techniques," *Proc. 33-th Design Automation Conference*, pp. 258-261 (June 1996).

[12] J. Jain, R. Mukherjee, and M. Fujita, "Advanced Verification Techniques Based on Learning," *Proc. 32-th ACM/IEEE Design Automation Conference*, pp. 420-426 (June 1995).

[13] W. Kunz, "HANNIBAL: An Efficient Tool for Logic Verification Based on Recursive Learning," *Proc. Int'l Conf. on CAD*, pp. 538-543 (Nov. 1993).

[14] C. E. Leiserson and J. B. Laxe, "Retiming Synchronous Circuitry," In *Algorithmica*, 6(1) (1991).

[15] Y. Matsunaga, "An Efficient Equivalence Checker for Combinational Circuits," *Proc. ACM/IEEE Design Automation Conference*, pp. 429-634 (June 1996).

[16] C. Pixley, "A Theory and Implementation of Sequential Hardware Equivalence", *IEEE Trans. on CAD*, pp. 1469-1494, (Dec. 1992).

[17] C. Pixley, V. Singhal, A. Aziz, and R. K. Brayton, "Multi-level Synthesis for Safe Replaceability," *Proc. Int'l Conf. on CAD*, pp. 442-449 (Nov. 1994).

[18] S. M. Reddy, W. Kunz, and D. K. Pradhan, "Novel Verification Framework Combining Structural and OBDD Methods in a Synthesis Environment," *Proc. 32-th ACM/IEEE Design Automation Conference*, pp. 414-419 (June 1995).

[19] R. Rudell, "Dynamic Variable Ordering for Ordered Binary Decision Diagram," *Proc. Int'l Conf. on CAD*, pp. 42-47 (Nov. 1993).

[20] N. Shenoy and R. Rudell, "Efficient Implementation of Retiming," *Proc. Int'l Conf. on CAD*, pp. 226-233 (Nov. 1994).

[21] H. J. Touati, H. Sarvoj, B. Lin, R. K. Brayton, and A. Sangiovanni-Vincentelli, "Implicit State Enumeration of Finite State Machines Using BDD's," *Proc. Int'l Conf. on CAD*, pp. 130-133 (Nov. 1990).

[22] "SIS: A System for Sequential Circuit Synthesis," Report M92/41, University of California, Berkeley (May 1992).