AQUILA: An Equivalence Checking System for Large Sequential Designs

Shi-Yu Huang, Kwang-Ting Cheng, Kuang-Chien Chen,
Chung-Yang Huang, and Forrest Brewer, *Member*, *IEEE*

Abstract—In this paper, we present a practical method for verifying the functional equivalence of two synchronous sequential designs. This tool is based on our earlier framework that uses Automatic Test Pattern Generation (ATPG) techniques for verification. By exploring the structural similarity between the two designs under verification, the complexity can be reduced substantially. We enhance our framework by three innovative features. First, we develop a local BDD-based technique which constructs Binary Decision Diagram (BDD) in terms of some internal signals, for identifying equivalent signal pairs. Second, we incorporate a technique called partial justification to explore not only combinational similarity, but also sequential similarity. This is particularly important when the two designs have a different number of flip-flops. Third, we extend our gate-to-gate equivalence checker for RTL-to-gate verification. Two major issues are considered in this extension: 1) how to model and utilize the external don't care information for verification; and 2) how to extract a subset of unreachable states to speed up the verification process. Compared with existing approaches based on symbolic Finite State Machine (FSM) traversal techniques, our approach is less vulnerable to the memory explosion problem and, therefore, is more suitable for a lot of real-life designs. Experimental results of verifying designs with hundreds of flip-flops will be presented to demonstrate the effectiveness of this approach.

Index Terms—Design verification, formal verification, equivalence checking, state exploration.

1 Introduction

Equivalence verification involves checking if the implementation of a design is functionally equivalent to an earlier version which is described at the same level of abstraction, or to a specification described at a higher level of abstraction in the design hierarchy. 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 [7], [10], [36]. Even though these transformations are correct in theory, 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, testability, or power dissipation requirements in the late design cycle, verification is even more important. The problem of comparing the functions of two gate-level sequential circuits has been studied extensively during the past few years [11], [39], [9], [13], [2]. In these approaches, the circuits are regarded as finite state machines and characterized by a transition relation and a set of output functions using Binary Decision Diagrams (BDDs) [6], [35]. 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 proven by checking if the function of the product machine's every primary output is tautology "0." These algorithms are quite efficient for small to medium-sized circuits. However, they can easily fail on larger designs due to the memory explosion problem.

Exploring the structural similarity between the two circuits under verification has been shown to be effective for reducing the complexity of verifying combinational circuits [4], [5], [22], [34], [20], [18], [30], [33]. The key idea in these algorithms is that, instead of directly examining the functional equivalence of the primary outputs, equivalent internal signal pairs are first identified. This process proceeds forward from the primary inputs toward the primary outputs. Whenever an internal equivalent signal pair is identified, the two signals in this pair are merged together for speeding up the subsequent verification process [5]. This type of equivalence checking algorithm is referred to as "incremental verification" in the sequel. It significantly reduces the time and space complexity of equivalence checking, and thus, is more suitable for large designs.

We have extended this idea to sequential circuits [16]. In this extension, sequential Automatic Test Pattern Generation (ATPG) techniques [8], [13], [1] are used to identify the equivalent flip-flop pairs and internal signal pairs. A computational model called *miter* [13], [5] is constructed and then the sequential backward justification technique is

- *S.Y. Huang is with the Department of Electrical Engineering, National Tsing-Hua University, HsinChu, Taiwan. E-mail: syhuang@ee.nthu.edu.tw.*
- *K.-T. Cheng, C.-Y. Huang, and F. Brewer are with the Department of Electrical and Computer Engineering, University of California, Santa Barbara, CA 93106. E-mail: {timcheng, ric, forrest}@ece.ucsb.edu.*
- *K.-C. chen is with Verplex Systems Inc., Santa Clara, CA. E-mail: kchen@verplex.com.*

Manuscript received 1 June 1997; revised 1 June 1998; accepted 1 July 1999. For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 111799.

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 subproblems: verifying the equivalence of a number of candidate flip-flop pairs, internal signal pairs, and then primary output pairs. But, because the search space for each subproblem is still very high, we further developed several techniques to cut down the search space. Furthermore, a preprocessing algorithm was also developed particularly for retimed circuits to reduce the time and space complexity [17].

The efficiency of the incremental verification relies on the techniques for identifying the equivalent internal signal pairs. An observation shows that a high percentage of equivalent pairs can be identified by only considering a small subcircuit surrounding the candidate signal pair. In this paper, we first describe an algorithm for identifying these signal pairs for combinational circuits via local BDDs. Here, the supporting variables of a local BDD could be some selected internal signals, instead of primary inputs. This enhancement [18] makes our approach less sensitive to the circuit's structural similarity than the pure ATPG-based techniques [5], [22]. A similar idea was also independently developed at the same time by Matsunaga in [30]. Second, we extend this local BDD-based engine for sequential circuits by a technique called *partial justification*. This technique is particularly essential as verifying two large designs with a different number of flip-flops.

Based on this robust gate-to-gate equivalence checker, we further extend it for verifying a gate-level implementation against its structural RTL specification. This RTL-to-gate verification is done in two steps. First, we perform the fast-path synthesis (i.e., synthesis/optimization with low effort) on the RTL specification to derive a gate-level specification. Then, we compare the gate-level specification with the gate-level implementation. One major problem with this approach is the *false negative* problem—the verifier may report inequivalence, even though the gate-level implementation correctly realizes the specified behavior of the RTL specification. The false negative problem is possible because an RTL design could be incompletely specified (e.g., dangling else statement). We refer to an input sequence as an *external don't care* (sequence) if the output response of this sequence is not defined in the RTL specification. Otherwise, it is called a *care* input sequence. External don't cares could be used for design optimization [3]. Different interpretation of these external don't cares may result in different logic implementations. To resolve this issue, we first characterize the don't care conditions as a sequential single-output network [3]. Then, during the process of gate-to-gate verification, we can take these don't cares into account to avoid the false negative problem.

Incremental verification is particularly efficient for verifying two sequential circuits with very similar encodings. However, the approach is not advantageous for verifying two circuits with different encodings, e.g., a minimal-bit encoded circuit and an one-hot encoded circuit. To circumvent this limitation, we combine the incremental verification with the symbolic finite state machine (FSM) traversal techniques. The idea is to implicitly partition the miter into two portions: 1) a portion in which the specification and the implementation share similar encodings and 2) a portion in which the encodings differ. Then, specific techniques can be applied to each portion. For instance, consider a processor-like design with a data-path and a controller. During the logic synthesis and optimization process, the data-path portion, containing functional units, registers, counters, shift-registers, etc., are rarely reencoded. While the encoding of the controller portion might be changed. In this unified method, we first perform FSM traversal on the differently encoded portions to derive a subset of unreachable states. These precomputed unreachable states are then used to assist the following incremental verification process. By integrating FSM traversal and incremental verification together, we are more likely to verify designs that are beyond the capability of either one of the two approaches alone.

The rest of this paper is organized as follows: Section 2 reviews our earlier framework. Section 3 describes a local BDD-based technique for combinational verification. Section 4 generalizes it to sequential circuits based on a technique called partial justification. Section 5 discusses the extension for RTL-to-gate verification. In Section 6, we presents the experimental results. In Section 7, we conclude.

2 Preliminary

2.1 The 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 known reset state externally, different definitions of sequential equivalence, such as *hardware sequential equivalence* [31], *safe replaceability* [32], [37], and *3-valued safe replaceability* [16] can be used. The comparisons of these definitions are beyond the scope of this paper and referred to [16]. We assume both circuits under verification have an external reset state in this paper, but the discussion can be applied to checking 3-valued safe replaceability or hardware sequential equivalence if a synchronization sequence for both *circuits under verification* (CUVs) is provided [31]. The verification is performed on the computational model called *miter* as shown in Fig. 1. For simplicity without losing generality, we assume C_1 and C_2 are both single output circuits and their outputs are o_1 and o_2, respectively. The 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. The problem is to decide if the responses of o_1 and o_2 are always identical for every possible input sequence.

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.

Definition 2 (Equivalent pair). (a_1, a_2) *is called an equivalent pair if the binary values of signal* a_1 *and* a_2 *in response to any care input sequence are identical.*

Definition 3 (Complementary pair). (a_1, a_2) *is called a complementary pair if the binary values of signal* a_1 *and* a_2 *in*

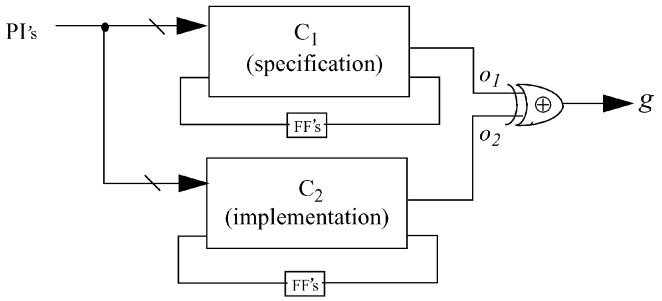


Fig. 1. The verification model for circuits C_1 and C_2.

response to any care input sequence are complementary to each other.

Definition 4 (Distinguishing sequence). *A care input sequence D is a distinguishing sequence for a signal pair (a_1, a_2) if applying D can produce value combination $(0,1)$ or $(1,0)$ at a_1 and a_2.*

2.2 The ATPG-Based Framework

We review the procedure of using a sequential ATPG program for verification in this subsection. We first build the miter of the two circuits. Then, a modified ATPG is used to search for a distinguishing sequence for primary output pair (o_1, o_2), which is also a test sequence for g stuck-at-0 fault. The search process is performed on the iterative array model using the reverse-time processing techniques, as shown in Fig. 2. Since we are dealing with an output fault, the forward fault-effect propagation is not necessary. Hence, the search process consists of only two stages: 1) fault injection, and 2) sequential backward justification.

During the process of sequential backward justification, value assignments at the present state lines of each time-frame are 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., $(000 \mid 000)$, is contained in a newly generated state requirement, then it indicates that a test sequence has been found and the justification process terminates. In the example of Fig. 2, $T = (v_3 v_2 v_1)$ 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 three 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 (i.e., no input sequence can bring the miter into a state contained by the state-cube). 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 two CUVs for this purpose. This procedure has two major steps: 1) identify equivalent flip-flop pairs, and 2) identify equivalent internal pairs. The first step is an iterative process until some stable condition is reached. In the subsection, we review this iterative algorithm and explain why it is more efficient than directly finding equivalent flip-flop pairs. The overall algorithm of our ATPG-based framework will be shown after that.

2.3 An Inductive Algorithm for Identifying Equivalent FF-Pairs

We proposed the notion of *assume-and-then-verify* to identify equivalent flip-flop pairs more efficiently in [16]. A similar, but more restricted algorithm was independently proposed in [12]. In our algorithm, we first derive an initial *candidate* equivalent flip-flop pairs (or equivalent groups) by simulating a large number of functional patterns, or by name comparison if possible. Two flip-flops or signals are considered as a candidate pair if their responses to the applied functional sequences 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 *not* equivalent). At the end of this process, every survivor flip-flop pair is guaranteed to be equivalent. This algorithm is referred as inductive because we rely on induction to prove its correctness. The details of the proof are provided in [16]. In Section 4, we will further generalize this algorithm in order to apply local-BDD-based techniques to sequential equivalence checking.

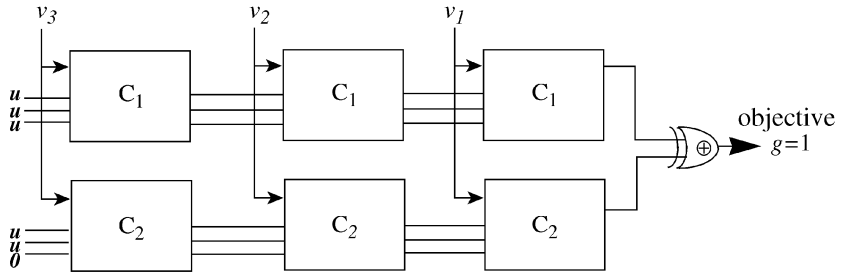


Fig. 2. An example of backward justification to find a distinguishing sequence.

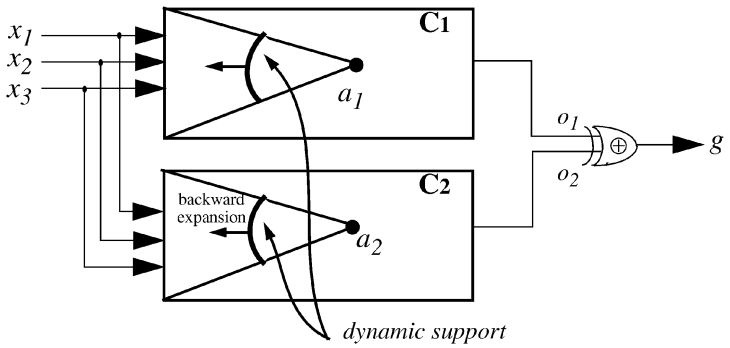


Fig. 3. The dynamic support that expands toward the primary inputs for verifying the equivalence of (a_1, a_2).

3 Combinational Equivalence Checking Using Local BDDs

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 BDDs*. In this section, we consider only combinational verification. In the next section, we generalize it for sequential circuits.

3.1 A Sufficient Condition for Equivalent Signal Pair

We first define the discrepancy function and then show a sufficient condition for checking the equivalence of a signal pair.

Definition 5 (Discrepancy function). *An input vector v is a distinguishing vector for a signal pair (a_1, a_2) if the application of v can produce $(0,1)$ or $(1,0)$ at a_1 and a_2. The characteristic function of the set of distinguishing vectors is called a* discrepancy function *and denoted as $Disc(a_1, a_2)$. If the functions of a_1 and a_2 are denoted as $\mathrm{F}(a_1)$ and $\mathrm{F}(a_2)$, respectively, then $Disc(a_1, a_2) = \mathrm{F}(a_1) \oplus \mathrm{F}(a_2)$.*

For larger designs, it is not feasible to construct the global BDD of the discrepancy for a signal pair. In [5], an ATPG technique is applied to trade time for space. Although it is quite efficient for circuits with significant structural similarity (e.g., circuits after combinational redundancy removal), in our experience, it might be very time-consuming or even inapplicable for circuits transformed extensively. Therefore, we propose another robust technique to compensate for ATPG-based approaches.

This technique is based on an observation that if there exists a cutset λ for the input cones of a_1 and a_2 such that no value combination of the cutset can produce $(0,1)$ or $(1,0)$ at (a_1, a_2), then (a_1, a_2) is an equivalent pair. We denote the discrepancy function according to the cutset λ for signal pair (a_1, a_2) as $Disc_\lambda(a_1, a_2)$. The following property states this sufficient condition of equivalence:

Property 1. *Signals a_1 and a_2 are equivalent if there exists a cutset λ for the input cones of a_1 and a_2 such that $Disc_\lambda(a_1, a_2) = \underline{0}$.*

3.2 Dynamic-Support for Constructing BDDs

Our experiments show that a large percentage of equivalent pairs satisfy the above sufficient condition without using the primary inputs as the cutset. Hence, we developed a heuristic that dynamically selects an appropriate cutset for each signal pair under equivalence checking. Since we only construct the discrepancy function based on a local cutset, we can handle much larger designs than the traditional approaches using global BDDs. Fig. 3 illustrates this dynamic expansion process of selecting the cutset. In the sequel, a cutset is also referred to as a *support*.

Example 1. Fig. 4 shows the snapshot of an example during the incremental verification process. We use the same label for a logic gate and its output signal. Suppose signal pairs (a, a'), (b, b'), and (c, c') have been proven to be equivalent pairs and signals a, b, and c have been replaced by their equivalent signals a', b', and c', respectively. In the sequel of this paper, signals like a', b', and c' are referred to as ***merged points*** as well. The algorithm proceeds to checking the equivalence of the primary output pair (o_1, o_2). Based on the cutset selection algorithm, we use $\lambda_1 = \{b', c'\}$ as the first level cutset, as shown in Fig. 4b. Then, we compute $Disc_{\lambda,1}(a_1, a_2)$, which is $\{(b', c') \mid (0,1)\}$. Since it is not empty, we further expand it backward to the second level cutset $\lambda_2 = \{x_1, a', x_3\}$. Then, we found that $Disc_{\lambda,2}(a_1, a_2)$ is a zero function. Hence, we conclude that (o_1, o_2) is equivalent and return. In this example, the distinguishing vector with respect to the first level cutset is proven to be *unsatisfiable* when the cutset just advances one level toward the primary inputs and the false negative problem is thus resolved efficiently.

In the worst case, we need to advance the frontier of the support all the way to the primary inputs to decide whether a signal pair is equivalent. This requires the construction of global BDDs. To avoid this situation, we set a limit on the maximum levels of the backward support expansion process. If the signal pair cannot be proven equivalent after reaching the limit, we give up, treat them as inequivalent and move on to the next signal pair. The only exception is when the target signal pair is a primary output pair. Then, we switch to the ATPG technique to continue the search for any possible distinguishing vector. The routine of checking equivalence for a signal pair is shown in Fig. 5.

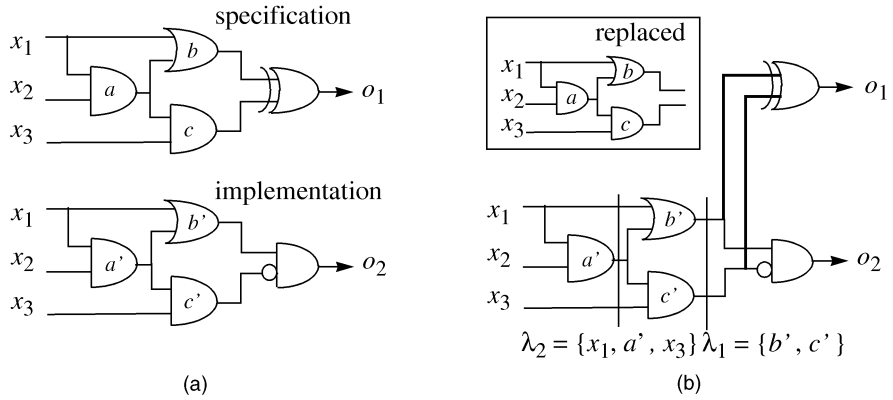


Fig. 4. A snapshot as checking the equivalence of signal pair (o_1, o_2). (a) Original circuits. (b) A snapshot after merging three pairs.

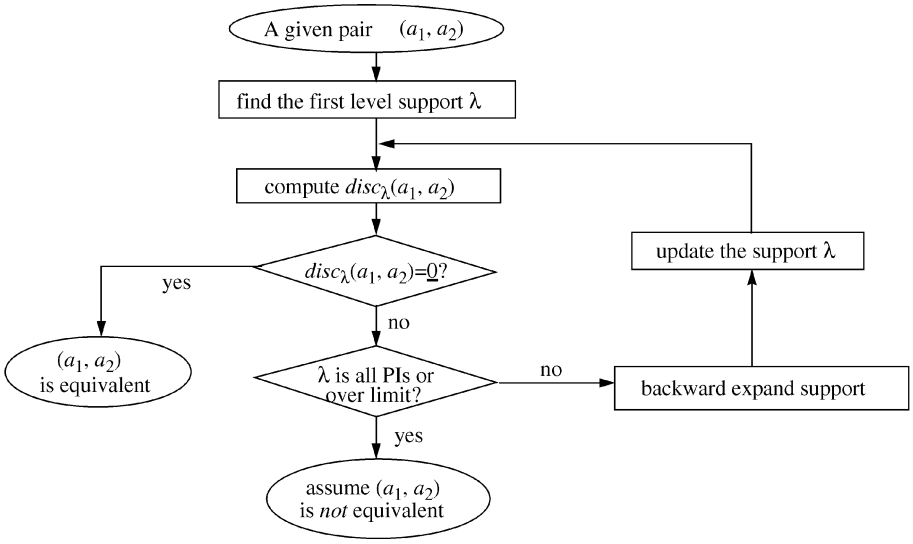


Fig. 5. Routine for checking combinational equivalence of a signal pair using local BDDs with dynamic support.

In our implementation, we expand the current support toward the primary inputs for two levels each time when we select the next support. Also, in an attempt to find a smaller cutset, we only select those equivalence points that have multiple fanouts as the supporting signals. The detailed pseudocode of this cutset selection algorithm[1] is shown in Fig. 6. This algorithm incorporates a recursive routine that perform a backward depth-first-traversal (towards the primary inputs) until merge points or primary inputs are reached.

4 Sequential Equivalence Checking

4.1 The Overall Algorithm for Sequential Verification

For simplicity, we assume that both sequential circuits under verification have an external reset state. The discussion can be generalized for circuits without a reset state based on the definitions of sequential hardware equivalence [31] or 3-valued safe replaceability [16]. Fig. 7 shows the overall flow of a sequential gate-to-gate equivalence checker that incorporates a hybrid approach employing both ATPG and local BDD techniques. In general, local BDDs are more efficient in proving the equivalence of an internal signal pair or a primary output pair, while ATPG is more effective in finding a counterexample if one exists. Combining the advantages of these two techniques makes this approach capable of handling many three phases in this algorithm:

Phase 1: Construct the miter and run simulation using a number of functional vectors to find the candidate equivalent flip-flop pairs and candidate internal signal pairs.

Phase 2: Simplify the miter in stages by identifying true equivalent flip-flop pairs and equivalent internal signals by a generalized inductive algorithm. This generalized algorithm consists of a two-level loop. The outer loop identifies equivalent flip-flop pairs and the inner loop identifies equivalent internal signal pairs. The details of this algorithm are discussed in the next subsection.

Phase 3: Check the equivalence of each primary output pair using a local BDD-based technique, followed by an ATPG-based technique, if necessary.

It is possible that, after functional simulation, several flip-flops in the specification and several flip-flops in the

1. Another cutset selection algorithm proposed by Matsunaga can be found in [30].

```
find_next_cutset (T)
    T: target signal pairs or current cutset.
{
    CUT= ϕ;          // next cutset set (to be returned)
    VISITED = ϕ;     // a hash table of nodes been visited during the traversal
    signal;          // a signal under consideration
    fanin;           // a fanin signal of a target signal

    foreach_signal_in_target_set(T, signal){
        VISITED = VISITED ∪ {signal};
        foreach_fanin(signal, fanin){
            recursive_dfs_traversal(fanin, VISITED, CUT);
        }
    }
    return(CUT);
}

recursive_dfs_traversal(target_signal, VISITED, CUT)
{
    if(target_signal is in VISITED);/* early return */
    else{
        VISITED = VISITED ∪ {target_signal}; /* remember visited signals */
        if(target_signal == PI or MERGE_POINT_with_multiple_fanouts){
            CUT = CUT ∪ {target_signal}; /* found a new cutset signal */
        }
        else{
            foreach_fanin(target_signal, fanin){
                recursive_dfs_traversal(fanin, VISITED, CUT);
            }
        }
    }
    return;
}
```

Fig. 6. The pseudocode for cutset selection.

implementation are all indistinguishable. In this kind of situation, all these flip-flops can be regarded as an equivalence class and merged together in the reduced model, and the monotone filtering process can be generalized as an iterative refinement process of the equivalence classes. Each iteration may further split an equivalence class into subsets.

The rest of this section is organized as follows: Section 4.2 discusses the ideas of exploring sequential similarity between the two CUVs by an inductive algorithm generalized from the one proposed in [16]. Section 4.3 discusses the computational details of checking the sequential equivalence of a signal pair using local BDDs within this framework.

4.2 Two-Level Inductive Algorithm

As mentioned in Section 2, we use the concept of *assume-and-then-verify* to identify the equivalent flip-flop pairs. This algorithm allows equivalence checking to be performed on a simplified miter, instead of the original miter, and thus, can identify equivalent flip-flop pairs much more efficiently. For the cases when the design has a reset state, Fig. 8 shows an example of the simplified model with $\{(y_1, z_1), (y_2, z_2)\}$ being regarded as the candidate flip-flop pairs. As opposed to the algorithm described in [16], where we only impose the equivalence constraint on each candidate PS-pair, here we actually merge them (or, more precisely, replace y_1 and y_2 by z_1 and z_2, respectively). We call this model the *first-level reduced model*.

It is worth mentioning that the efficiency of this algorithm depends on the underlying techniques of equivalence checking. To demonstrate this point, consider an extreme case in which the two designs have a perfect one-to-one flip-flop correspondence and are indeed combinationally equivalent. In the first-level reduced model, it needs to be proven that each candidate NS-pair is equivalent. If the traditional forward FSM traversal is used, then this process still requires the exploration of the entire state space of the design. On the other hand, if the backward justification techniques are employed, then only one time frame is sufficient to prove the equivalence for each candidate NS-pair. The difference in computational complexity between these two is huge. There exist many cases where state traversal cannot be completed, while the BDD-representation of the combinational portion can be easily constructed. This example indicates that the advantage of this speed-up technique can be fully exploited only if it is accompanied with the backward justification techniques.

4.2.1 The Second Level Assume-and-then-Verify

The process of proving the equivalence of a candidate flip-flop pair can be further sped up by identifying equivalent internal signal pairs in the first-level reduced model.

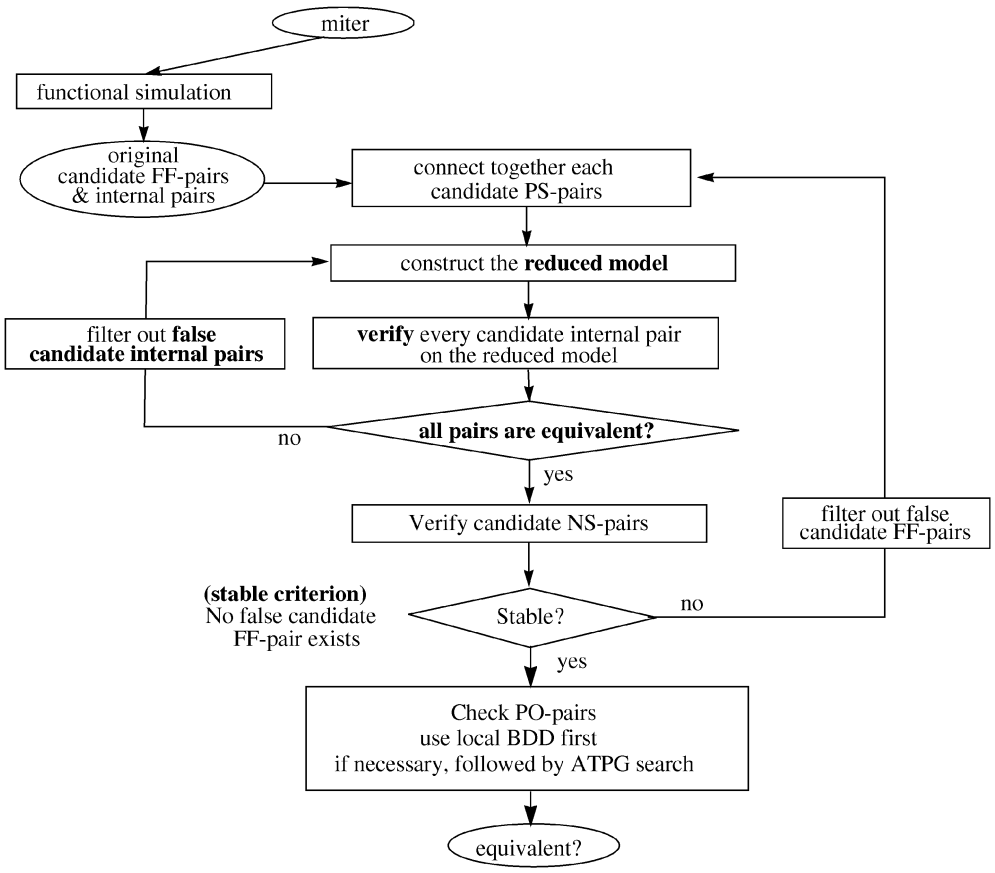


Fig. 7. Incremental verification flow for sequential circuits using local BDD and ATPG.

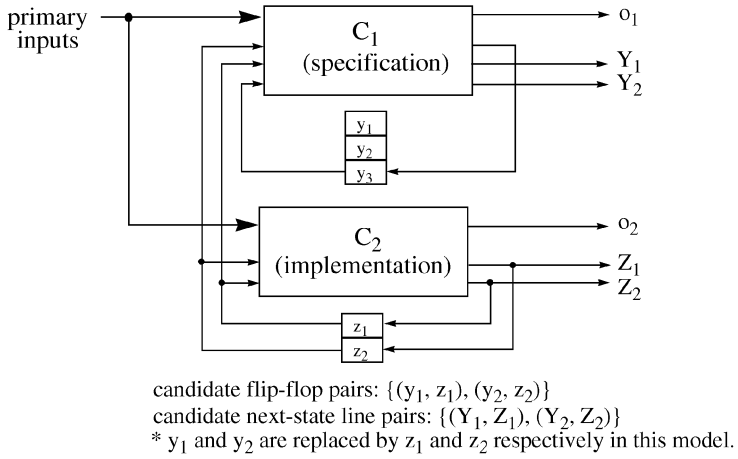


Fig. 8. The first-level reduced model for exploring sequential similarity.

Consider the example shown in Fig. 9a. In addition to the candidate flip-flop pairs, suppose we have three candidate internal pairs, $\{(a, a'), (b, b'), (c, c')\}$, derived from the results of the preprocessing stage.

Similar to the identification of equivalent flip-flop pairs, we incorporate a monotone filter in this generalization. Initially, we assume that all internal signal pairs are equivalent. Then, we iteratively identify false candidate internal pairs (as will be defined later) and remove them from the candidate list. This process continues until no false candidate pair exists.

Identifying false candidate pairs is an iterative process as well. Each iteration involves two steps:

Step 1: Construct a *second-level reduced model* that merges the candidate PS-pairs as well as the candidate internal pairs (Fig. 9a). Also, a copy of the combinational portion of the first-level reduced model, called *assumption checker*, is created and attached to the reduced model as shown in Fig. 9b.

Step 2: Check the equivalence of each candidate internal pair in the assumption checker in stages. A candidate

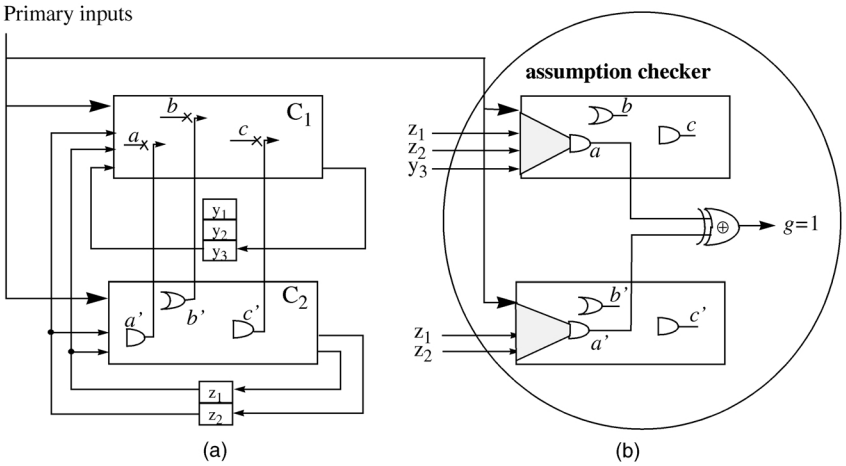


Fig. 9. (a) Second-level reduced model and (b) the assumption checker.

pair (a_1, a_2) is *false* if there exists an input sequence that differentiates (a_1, a_2) in the assumption checker network. Fig. 9b shows a snapshot during the checking of the equivalence of signal pair (a_1, a_2). The computational details will be described in the next section. If no false candidate pair exists, then the process reaches a stable condition and exits. Otherwise, the process starts over with a smaller set of candidate internal pairs after excluding the identified false candidate pairs.

When the iterative process reaches a stable condition, it can be proven by induction that each survivor candidate pair is indeed equivalent.

Property 5.1. *If an input sequence applied to the first-level reduced model does not differentiate any candidate internal pair, then it will bring the first-level and second-level reduced models to the same final state.*

Lemma 1. *Every candidate internal pair remaining in the candidate list is indeed equivalent in the first-level reduced model after the stable condition of the above two-step process is reached.*

Proof. We prove that the survivor candidate pair is equivalent for any input sequence by induction. The induction is on the length of the input sequence.

(I. BASIS): For any input sequence of length "1," every candidate pair is equivalent. Otherwise, some candidate pairs in the assumption checker network would have been classified as false pairs.

(II. INDUCTION-STEP): Assume that every candidate internal pair is equivalent for any input sequence of length n in the first-level model. From Property 5.1, it follows that the set of reachable states of the first-level and second-level reduced models in n clock cycles are the same. As a result, if we cannot differentiate any candidate pair in the assumption checker network of the second-level reduced model in $n+1$ cycles, then we cannot differentiate any pair in the first-level reduced model either. Therefore, every candidate internal pair is equivalent for any input sequence of length $n+1$ once the stable condition is satisfied.

(III. CONCLUSION) Based on the above inductive analysis, we conclude that every candidate internal pair that survives the filtering process is indeed equivalent *for any input sequence* in the first-level reduced model. □

The above lemma can also be interpreted as, *if there exists at least one false pair, then the suggested algorithm will identify at least one pair (i.e., the process will not be stabilized)*. Fig. 10 shows the pseudocode of AQUILA.

4.3 Symbolic Backward Justification

In this section, we describe the details of checking the sequential equivalence of a signal pair (a_1, a_2) using local BDDs. Computationally, this is performed on the iterative array model. Contrary to an ATPG-based procedure, the backward justification is now performed by a sequence of symbolic preimage computations.

At the last time frame of this procedure, the two signals in the target pair (a_1, a_2) are treated as pseudoprimary outputs and the set of input vectors at primary inputs (PIs) and present state lines (PSs) that can differentiate this target pair is computed. Fig. 11 shows an illustration. The characteristic function of this set is called the discrepancy function at time frame 0, denoted as $Disc^0(a_1, a_2)$. In the rest of this section, the superscript of a notation indicates the index of the associated time frame. The index of the last time frame is 0 and increases in a backward manner, as shown in Fig. 11. After existentially quantifying out all primary inputs of $Disc^0(a_1, a_2)$, we arrive at 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). Then, the checking of the equivalence of (a_1, a_2) is reduced to a sequential >backward justification process that decides if there exists an input sequence that will bring the reduced miter from the reset state to any state in $SR^0(a_1, a_2)$.

```
AQUILA (M1, M2, Pf, Pi)
        Pf = candidate equivalent flip-flop pairs;
        Pi = candidate internal pairs sorted from PI to PO;
{
    Let  A = assumed sequentially equivalent signal pairs;
        Ff = false flip-flop pairs;
        Fi = false candidate internal pairs;

    while(1) { // first level loop
        construct the first-level reduced model;

        /*--- Step 1: identify equivalent internal signal pairs ---*/
        while(1){ // second level loop
            /*--- simplify miter and decide candidate pairs to be assumed ---*/
            set A = ϕ;
            foreach internal pair p = (a1, a2) in Pi {
                check if equivalence for a single vector by local BDD;
                if (equivalent) replace a1 by a2;
                else if (this pair is critical according to a heuristic) {
                    A = A + p;
                    replace a1 by a2; // assume sequentially equivalent
                }
            }
            /*--- check assumption ---*/
            construct the 2nd-level reduced model with assumption checker network;
            Fi = ϕ;
            foreach p = (a1, a2) in A {
                check sequential equivalence of (a1, a2) by partial justification;
                if( ! equivalent ) { Fi = Fi + p; }
            }
            if (Fi != ϕ) Pi = (Pi - Fi ); // remove false candidate internal pairs
            else break; // no false candidate internal pair, exit 2nd loop;
        }
        /*--- Step 2: check the equivalence of each next-state pair ---*/
        Ff = ϕ;
        foreach NS-pair p in Pf {
            check equivalence of p;
            if( ! equivalent ) {Ff = Ff + p; }
        }
        /*--- Step 3: check the stable condition ---*/
        if (Ff != ϕ) Pf = (Pf - Ff ); // remove false candidate flip-flop pairs
        else break; // no false candidate flip-flop pair, exit 1st loop;
    }
    /*--- Step 4: check the equivalence of primary output pair ---*/
    Apply local BDD followed by ATPG search; report any inequivalent PO-pair;
}
```

Fig. 10. The pseudocode of AQUILA.

A symbolic backward justification process consists of a sequence of preimage computations and existential quantifications. At each time frame i, $(i > 0)$, $SR^i(a_1, a_2)$ is derived by first computing the preimage of $SR^{(i-1)}(a_1, a_2)$, followed by existentially quantifying out every primary input. The symbolic techniques for preimage computation [11], [39], [9] can be directly applied. The stopping criteria of this backward justification process will be checked after $SR^i(a_1, a_2)$ is derived at time frame i to see if another backward time frame expansion is needed:

1. justified criterion: when the reset state is contained in $SR^i(a_1, a_2)$,
2. unjustifiable criterion: when $SR^i(a_1, a_2)$ is a zero function,
3. fixed-point (or cyclic) criterion: when $SR^i(a_1, a_2)$ does not contain the reset state, but is contained in the union of the state requirements derived so far,

$$\sum_{k=0}^{i-1} SR^k(a_1, a_2).$$

If the unjustifiable criterion or the fixed-point criterion is met, then the target signal pair is an equivalent pair. On the other hand, if the justified criterion is encountered, then it is

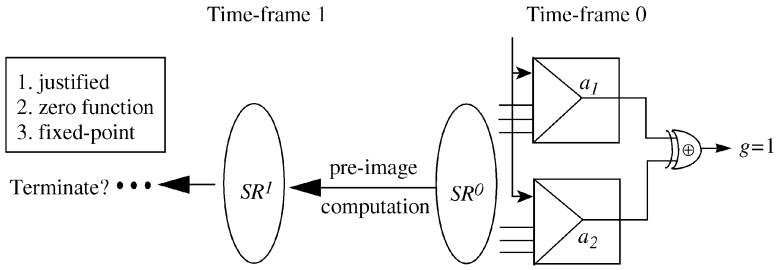


Fig. 11. Illustrating a symbolic procedure for checking if $a_1 = a_2$.

```
Symbolic-equivalence-checking (a1, a2)
{
    compute discrepancy function Disc0(a1, a2) at the last time frame;
    derive the state requirement SR0(a1, a2) by doing an existential quantification on
PI  ;
    New = SR0(a1, a2);// new state requirements to be further justified
    SR = ϕ;                    // accumulated state requirements so far
    /*------ symbolic backward justification ------*/
    while(1) {
        c = check_stopping_criteria(New);
        switch(c){
            case JUSTIFIED: return(DIFFERENT);
            case UNJUSTIABLE: return(EQUIVALENT);
            case FIXED-POINT: return(EQUIVALENT);
            default: SR = SR ∪ New; // update accumulated state requirements
                     New = compute_preimage_and_existentially_quantify_PI   (SR);
                     break;
        }
    }
```

Fig. 12. The pseudocode for checking the sequential equivalence of a signal pair by symbolic backward traversal.

not an equivalent pair. The pseudocode of this process is provided in Fig. 12.

4.3.1 Partial Justification

The above symbolic backward justification may still cause a memory explosion for circuits with a large number of flip-flops, even though it is performed on the reduced model. Hence, a simple heuristic, called *partial justification*, is further used to enhance the approach. This heuristic can be used during the backward justification process for identifying equivalent flip-flop pairs, internal equivalent pairs, or for checking primary output equivalence.

During the verification process described in Fig. 7, assume that the identified equivalent pairs or assumed equivalent pairs in the fanins of (a_1, a_2) of the reduced model 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 PIs and PSs directly, we select a local cutset, denoted as λ_0, in the pair's fanins. We use $Disc^0_{\lambda_0}(a_1, a_2)$ to denote the characteristic function of the set of value combinations at λ_0 that can differentiate (a_1, a_2). The cutset signals in λ_0 should be either previously identified *merge points* (defined in Example 1) or primary inputs. Using the heuristic of *dynamic support* described in an earlier section, expanding the cutset toward the PIs may increase the success rate in proving the equivalence of the target pair. Suppose the cutset has been pushed back toward the PIs and PSs for a number of logic levels within >the same time frame, but (a_1, a_2) still cannot be proven equivalent, i.e., $Disc^0_{\lambda_0}(a_1, a_2)$ is not a zero function. At this point, the cutset λ_0 may contain some present state lines, some primary inputs, and some internal signals. We then do an existential quantification on the signals in λ_0 that are not present state lines from $Disc^0_{\lambda_0}(a_1, a_2)$ to obtain a *superset* of the state requirement function, denoted as $SR^0_{\lambda_0}(a_1, a_2)$. This function characterizes a necessary condition that should be satisfied at the present state lines in order 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 stopping criteria is satisfied. For simplicity, we assume only one cutset is selected at each time frame i, denoted as λ_i. The *overestimated* state requirement function at time frame i, $SR^i_{\lambda_i}(a_1, a_2)$, can be derived from $SR^{i-1}_{\lambda_{i-1}}(a_1, a_2)$ by preimage computation and existential quantification on the non-PS supporting signals in λ_i. After this partial justification process, if the overestimated state requirements for differentiating (a_1, a_2) satisfy the unjustifiable or fixed-point criterion, then it can be proven that (a_1, a_2) is equivalent. However, if the overestimated state requirements are justified (i.e., reachable from the reset state), the target signal pair may not be indeed inequivalent unless we use only the PIs and PSs as the cutset, in which

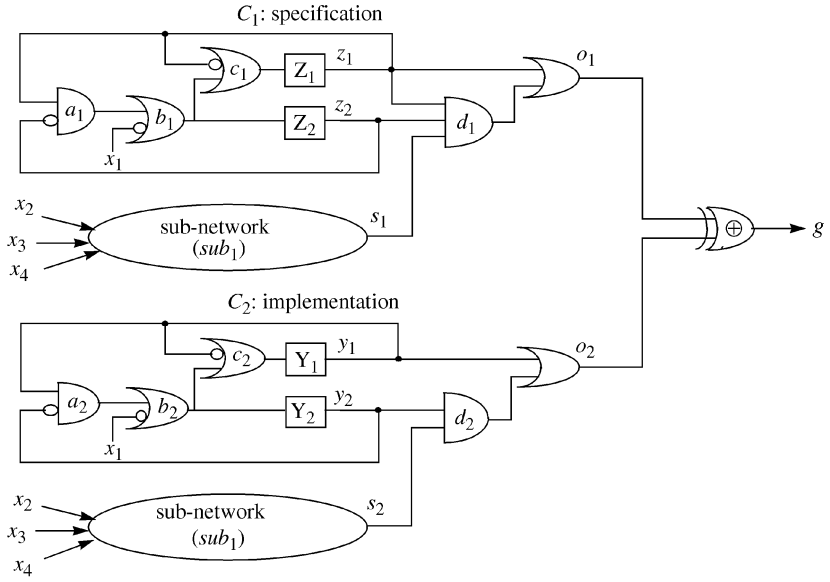


Fig. 13. An example for gate-to-gate equivalence checking.

case the partial justification degenerates to a complete justification.

Selecting a good cutset is essential for this heuristic. Similar to the combinational cases, we expand the cutset dynamically from the target pairs toward the primary inputs and he present state lines within each time frame. If this heuristic cannot conclude whether or not a pair is equivalent, we take the following actions. If the target pair consists of internal signals, we pessimistically assume that they are not equivalent in order not to run into the memory explosion which occurrs as we further expand the cutset. Note that, due to this early abortion, the *false negative* problem may arise. That is, some equivalent signal pairs might be overlooked. But, since the identification of the internal equivalent signal pairs is only a speed-up technique, failure to identify certain equivalent signals pairs only adds to the complexity of the equivalence checking at the primary output pairs. The final result of verification is not affected. If the target pair is a primary output pair, then we run ATPG as the heuristic of local BDDs fails to prove the equivalence or to find a distinguishing sequence. In this way, we switch to an algorithm that trades time for space as an effort to resolve the false negative problem.

4.4 An Example

Fig. 13 shows an example. For simplicity, we only apply one-level inductive algorithm. There are four primary inputs x_1, x_2, x_3, and x_4. Networks C_1 and C_2 contain subnetworks, sub_1 and sub_2, respectively. Suppose the outputs of these two subnetworks, s_1 and s_2, have been proven equivalent. In addition to these two subnetworks, C_1 (C_2) has two flip-flops denoted as Z_1 and Z_2 (Y_1 and Y_2), respectively. As for flip-flops, the outputs of Z_1 and Z_2 (Y_1 and Y_2) are present state lines and denoted as z_1 and z_2 (y_1 and y_2), respectively. The verification procedure is detailed as follows:

(Step 1): Perform simulation for a large number of random or functional sequences 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 equivalent flip-flop pairs.

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

(2.2) Verify the equivalence of $\{(Z_1, Y_1), (Z_2, Y_2)\}$. Sweep the miter from PIs toward POs to sequentially identify internal equivalent signal pairs. Signal pairs $\{(a_1, a_2), (b_1, b_2), (c_1, c_2)\}$ can be easily verified as equivalent one by one, and so can $\{(Z_1, Y_1), (Z_2, Y_2)\}$. Hence, we conclude that the assumption made in (2.1) is correct (i.e., (Z_1, Y_1) and (Z_2, Y_2) are indeed equivalent) and the process has stabilized.

(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 proven equivalent and, thus, we replace s_1 by s_2. The process moves on to check (d_1, d_2). Fig. 14 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 $\{(y_1, y_2, s_2) = (0, 1, 1)\}$ for differentiating d_1 and d_2. Let the characteristic function of this set be $Disc^0_\lambda(d_1, d_2)$. Since signal s_2 is not a present state line, it is existentially quantified from $Disc^9_\lambda(d_1, d_2)$ to derive the set of overestimated state requirements, which is $\{(y_1, y_2) = (0, 1)\}$. None of the three stopping criteria is satisfied. Hence, the backward justification process starts. At the next time frame, the preimage of $\{(Y_1, Y_2) \mid (0, 1)\}$ is an empty set and, thus, the state requirement $\{(y_1, y_2) = (0, 1)\}$ is unjustifiable. Signal pair (d_1, d_2) is sequentially equivalent and the two signals are merged together. The resulting circuit is shown in Fig. 15.

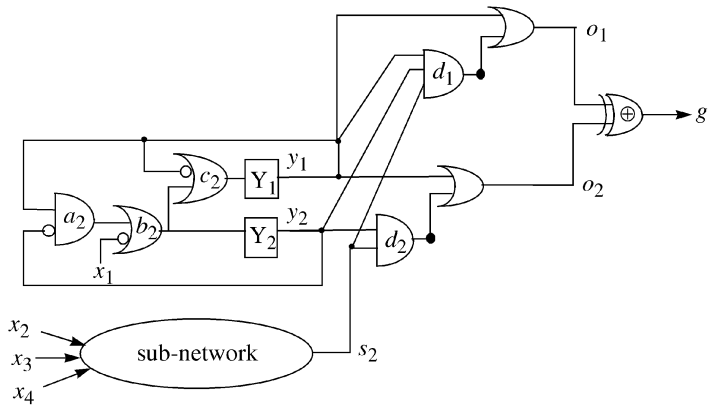


Fig. 14. A snapshot of miter before checking equivalence of (d_1, d_2).

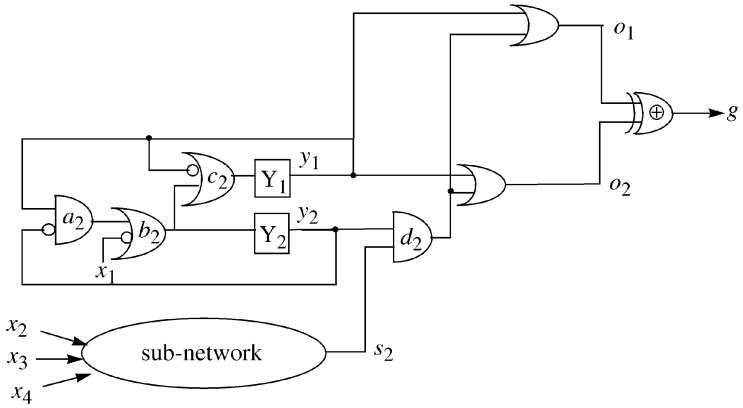


Fig. 15. A snapshot of miter after merging d_1 and d_2.

(3.2): Check output pair (o_1, o_2) using the local BDD. They can be proven equivalent by selecting $\{y_1, d_2\}$ as the cutset in Fig. 15.

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

5 Extension to RTL-to-Gate Verification

In this section, we discuss how to use a gate-to-gate equivalence checker for verifying an implementation against its structural RTL specification. We begin with the basic methodology and then discuss two key issues for this extension, i.e., external don't care modeling and integration of the symbolic FSM traversal with the incremental verification.

5.1 Don't Care Modeling

As mentioned in the introduction, checking if a gate-level implementation conforms to its RTL specification can be done in two steps: 1) fast-pass synthesis and then 2) gate-to-gate equivalence checking. In most cases, a design's RTL description is incompletely specified. Incomplete specification implies external don't care conditions that can be utilized for design optimization. Ignoring these external don't cares during verification may result in the false-negative problem. To resolve this problem, we generalize the computational model for verification, as shown in Fig. 16. In this generalized model, the external don't care set is represented as a sequential network with only one primary output dc. This don't care network is considered as part of the specification. It is not necessarily a network in terms of the primary inputs only, but could share flip-flops or even internal signals with the specification network. If an input sequence is a don't care sequence, then the don't care network produces a value "1" at signal dc. A value "0" indicates the applied input sequence is a care sequence. For example, a sequential multiplier may only produce one care output for every certain number of clock-cycles. Therefore, the don't care network could produce output response like 11111110, which means that only in the eighth cycle, the output is a care output. In this generalized miter, every output pair of the specification and the implementation is still connected to an exclusive-OR gate (denoted as g). This signal is further connected to a 2-input OR gate with signal dc as the other input. The output of this OR gate is denoted as signal $diff$.

Property 2. *Signal g stuck-at-0 fault in Fig. 16 is untestable if and only if signal o_1 is equivalent to o_2 for all care input sequences.*

An input sequence D should satisfy two conditions to be a care distinguishing sequence for (o_1, o_2): 1) D should differentiate (o_1, o_2), i.e., setting signal g to "1"; and 2) D should be a care sequence, i.e., setting signal dc to "0." These two conditions are exactly the same conditions as a test for signal g stuck-at-0 fault. This necessary and sufficient condition can be directly examined by a sequen-

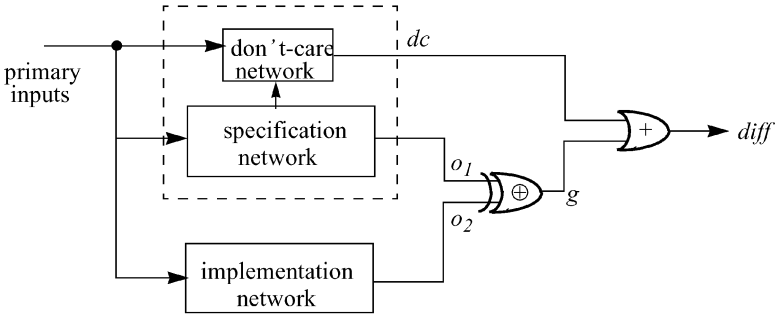


Fig. 16. Generalized miter considering external don't care set.

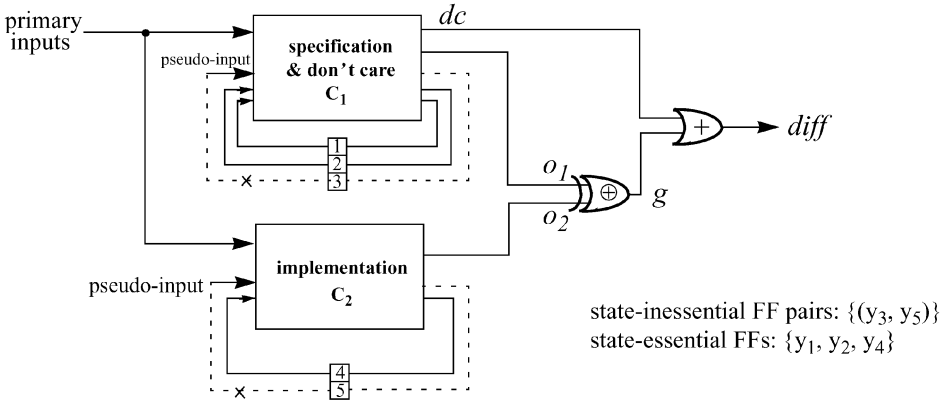


Fig. 17. An abstract miter for computing partial set of unreachable states.

tial ATPG program to prove the equivalence or to find a distinguishing sequence for a target signal pair. In addition, the idea of performing sequential verification *in stages* can still be applied to this generalized model.

Unlike external don't cares, internal don't cares are referred to the *impossible value combinations at some signals*, e.g., unreachable states. Internal don't cares are due to the temporal and/or spatial correlations among signals during the execution of the designs. In the context of design optimization, the external and internal don't cares need not be differentiated. However, for verification, they are different in essence. Unlike external don't cares, not considering internal don't cares does not cause any false negative problem. However, extracting these internal don't cares is essential in order to speed up the verification process in some cases. For example, if the backward ATPG search reaches an unreachable state, it can backtrack immediately to avoid unnecessary further search.

5.2 Combination of Forward FSM Traversal with Incremental Verification

Incremental verification and symbolic FSM-traversal based verification have different strengths and weaknesses. Incremental verification is effective for verifying designs with a great deal of internal similarity. FSM-traversal is structure-independent, while it is vulnerable to the memory explosion problem. In verifying a large RTL-synthesized circuit versus a custom optimized circuit which has different encoding for part of the design, either approach could have difficulties. For example, in a processor-like design, the specification and the implementation may have differently encoded controllers. Even though significant similarity exists in the data-path, the incremental verification approach may not succeed due to the dissimilarity in the controller and the FSM-traversal approaches could easily run into a memory explosion due to the huge state space mainly contributed by the data-path registers. In the following, we describe a method that combines the advantages of the above two approaches.

5.2.1 Computing a Partial Set of Unreachable States

In this subsection, we describe a procedure that performs FSM traversal on the *differently encoded portion of the miter* to compute a subset of unreachable states. First, we describe our heuristic for automatically deciding the differently encoded portion. Given a specification with a don't care network and an implementation, we first simulate a large number of functional input sequences to pair up the flip-flop pairs. A flip-flop is paired up with another flip-flop in the other network if the two flip-flops have identical responses for all applied functional care input sequences. The flip-flops which are paired are classified as *state-inessential* FFs. On the other hand, if a flip-flop is not paired, then it is classified as a *state-essential* FF. Fig. 17 shows an example of this classification. The specification and the don't care network has three flip-flops $\{y_1, y_2, y_3\}$, and the implementation has two flip-flops $\{y_4, y_5\}$. Suppose (y_3, y_5) is a candidate flip-flop pair after simulation. Then, the set of state-inessential FFs would be $\{y_3, y_5\}$ and the set of state-essential FFs would be $\{y_1, y_2, y_4\}$. Based on the above classification, we construct an *abstract miter*. This abstraction removes the state-inessential flip-flops and treats their outputs as pseudoinputs to the miter (illustrated in

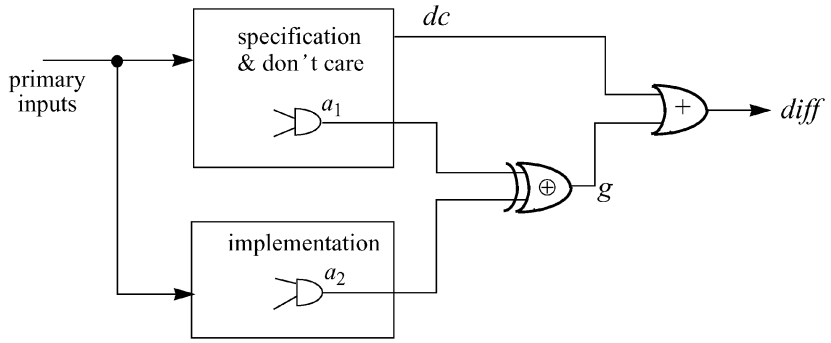


Fig. 18. Generalized miter for checking equivalence of (a_1, a_2) (a_1 is equivalent to a_2 if and only if g stuck-at-0 is untestable).

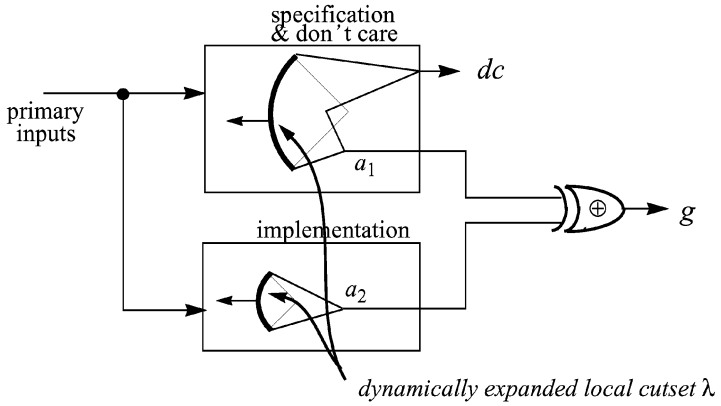


Fig. 19. Checking equivalence of (a_1, a_2) using local BDDs considering don't cares.

Fig. 17). Based on this abstract miter, we then perform FSM traversal to derive a set of unreachable states. These derived unreachable states are part of the internal don't care conditions, which can be utilized to speed up the subsequent incremental verification process.

We explain the process of computing a subset of unreachable states from the abstract miter using the example in Fig. 17. Since FSM traversal is a well-developed procedure, we refer the reader to the literature for the details. Suppose the primary input set is $\{x\}$, the pseudoinput set is $\{y_3, y_5\}$, and the set of present state lines is $\{y_1, y_2, y_4\}$. We denote the set of next state lines of the state-essential FFs as $\{\mathrm{Y}_1, \mathrm{Y}_2, \mathrm{Y}_4\}$ and their functions in terms of the present state lines, primary inputs, and pseudoinputs as $\{f_1, f_2, f_4\}$. Let the Boolean function of signal dc be F_{dc}. The transition relation of this abstract miter considering external don't care set can be described as a Boolean function:

$$Transition\text{-}relation = (\exists x\ \exists y_3\ \exists y_5)((Y_1 \otimes f_1) \cap (Y_2 \otimes f_2) \cap (Y_4 \otimes f_4) \cap \sim F_d c),$$

where $\exists$, $\cap$, $\otimes$ are the operators of existential quantification, AND, and exclusive-NOR. It can be seen that all the state-inessential FFs are existentially quantified in addition to the primary input variables. Also, the transition relation is intersected with the *care function* $(\sim F_{\mathrm{dc}})$ to guarantee the transition happens in the care space. Based on this transition relation, we can compute the next states of a given set of states in one shot (by image computation). A breadth-first traversal of the state transition graph from the reset state is employed to compute the reachable states of this abstract miter. We then perform a negation on this set to obtain the unreachable states. Note that the unreachable states derived in this way are also unreachable states for the original miter. Hence, we can use this information in the original miter to speed up the incremental verification process. For instance, if $(y_1, y_2, y_4) = (1, 1, 1)$ is a unreachable state in the abstract miter, then $(y_1, y_2, \mathbf{y_3}, y_4, \mathbf{y_5}) = (1, 1, -1, -)$ is a unreachable state cube in the original miter.

5.2.2 Incremental Verification Utilizing Don't Cares

Consider a target internal signal pair (a_1, a_2) under equivalence checking. The don't care network imposes a restriction on the input space of a distinguishing sequence for this pair. We discussed earlier how to use an existing sequential ATPG program to account for this restriction automatically (as shown in Fig. 18). In the following, we discuss how to generalize the idea of using local BDDs to check the equivalence of a signal pair under the don't care constraints. As illustrated in Fig. 19, a local cutset in the fanin cones of signal a_1, a_2, and dc is selected. Then, we construct the BDDs representing signal dc and signal g, denoted as $F_{\mathrm{dc}(\lambda)}$ and $F_{\mathrm{g}(\lambda)}$ in terms of the selected local cutset λ, respectively. We examine if $(F_{\mathrm{g}(\lambda)} \cap \sim F_{\mathrm{dc}(\lambda)})$ is the zero function to check the equivalence of (a_1, a_2). As long as $(F_{\mathrm{g}(\lambda)} \cap \sim F_{\mathrm{dc}(\lambda)})$ is the zero function, it is safe to claim that (a_1, a_2) is an equivalent pair.

Suppose F_{u} denotes the characteristic function of the unreachable states in terms of present state lines derived in the FSM traversal. We can use F_{u} to help prove the equivalence of a signal pair. Each time a local cutset λ is

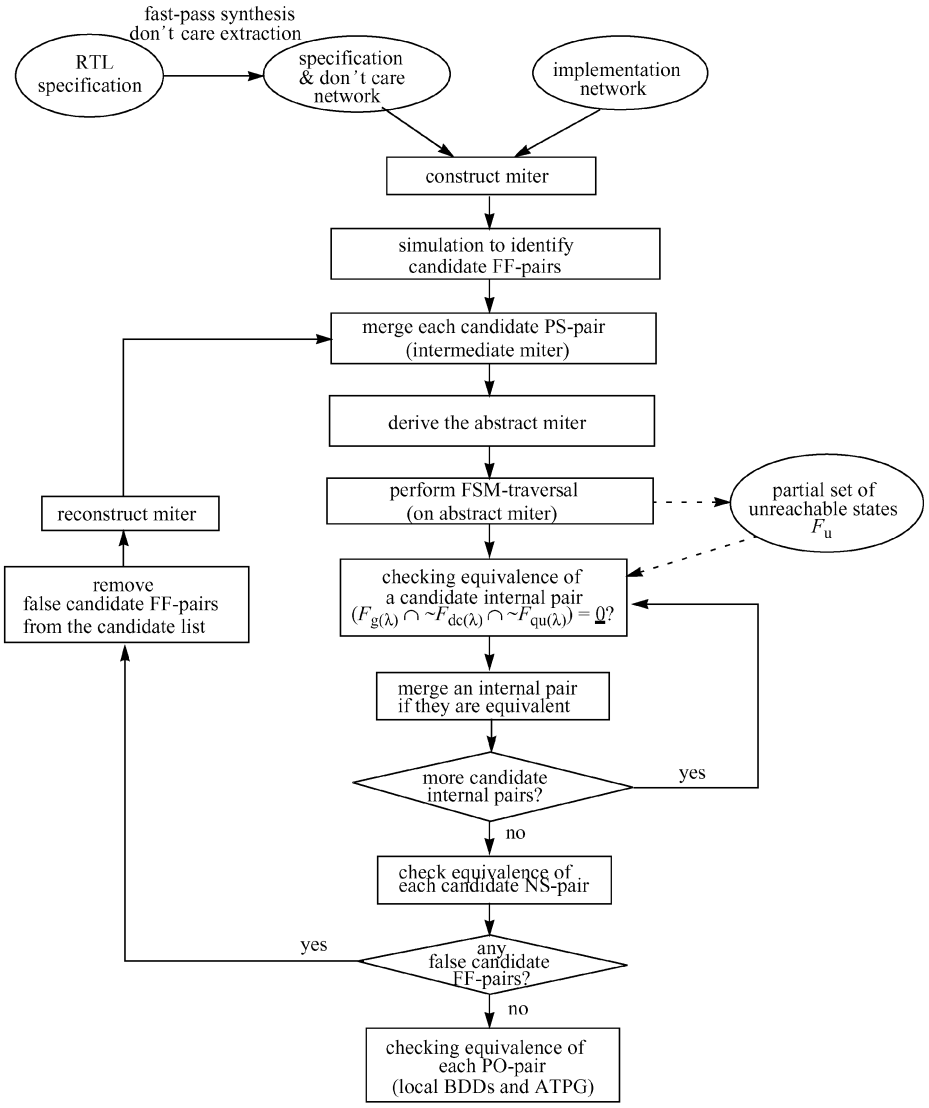


Fig. 20. Overall algorithm for RTL-to-gate verification.

selected, we universally quantify out those variables of the present state lines in F_u that are not in λ. Let the function of the quantified unreachable states be F_{qu}, which is a function exclusively in terms of λ. If $(F_{g(\lambda)} \cap \sim F_{dc(\lambda)} \cap \sim F_{qu(\lambda)})$ is the zero function, then the target pair is equivalent. Intuitively, the unreachable states, which represent the set of impossible value combinations at the present state lines, can be used to derive the impossible value combinations at the selected local cutset λ. To differentiate a signal pair, it is necessary to find a distinguishing vector in terms of λ. These distinguishing vectors in the care space are characterized by function $(F_{g(\lambda)} \cap \sim F_{dc(\lambda)})$. In addition, this vector should be reachable from the initial state. Function $(F_{g(\lambda)} \cap \sim F_{dc(\lambda)} \cap \sim F_{qu(\lambda)})$ represents the set of care distinguishing vectors that is reachable in terms of the local cutset λ. If this set is empty, then the target pair is equivalent.

5.3 The Overall RTL-to-Gate Verification Algorithm

We summarize our RTL-to-gate verification as follows: The RTL specification is first synthesized with low effort. External don't cares extracted from the specification are represented as a don't care network. The entire procedure involves three miter models: 1) the original model, (2) the intermediate model, and 3) the abstract model. The intermediate model is constructed by connecting each candidate PS-pairs together. On top of this intermediate model, the abstract model is then constructed temporarily by treating the input (output) of every candidate FF-pair as a pseudoprimary output (input) for computing a subset of unreachable states. The overall flow is shown in Fig. 20.

6 Experimental Results

6.1 Verify Combinational Transformations

We have implemented a prototype tool called AQUILA [19] for checking the equivalence of combinational or sequential designs in the SIS environment [38]. We integrate AQUILA with a sequential ATPG program *stg3* using the algorithm described in [8]. Table 1 shows the results of using our program to verify the correctness of ISCAS-85 benchmark circuits minimized by running *script.rugged* in SIS. These circuits are further mapped into the library cells defined in

TABLE 1
Results of Verifying Circuits Minimized by *script.rugged* and then Mapped into Library Cells Defined in *mcnc.genlib* in SIS

Circuit	# nodes of C1	# nodes of C2	# pairs with same name	# equivalent signal-pairs (equiv. / complement.)	AQUILA CPU-time
C432	123	97	1	43 / 17	5
C499	162	184	0	101 / 53	1
C880	302	182	0	57 / 23	1
C1355	474	184	0	106 / 61	2
C1908	441	214	7	148 / 77	6
C2670	694	298	2	119 / 82	15
C3540	956	560	3	169 / 179	28
C5315	1454	661	3	271 / 153	10
C6288	2353	1419	1	849 / 334	10
C7552	2118	881	3	418 / 245	21

TABLE 2
Results of Verifying ISCAS-85 Circuits after Redundancy Removal in SIS

Circuit	# nodes of C1	# nodes of C2	# equivalent signal-pairs (equiv. / complement.)	Reddy et. al. [34] CPU-time (sec)	Matsunaga [30] CPU-time(sec)	AQUILA CPU-time(sec)
C432	123	119	117 / 0	2	1	3
C499	162	162	144 / 0	2	1	1
C880	302	302	302 / 0	-	-	2
C1355	474	474	456 / 0	6	3	3
C1908	441	436	425 / 0	14	6	5
C2670	694	519	498 / 1	159	4	6
C3540	956	906	849 / 2	67	17	8
C5315	1454	1430	1401 / 0	372	14	10
C6288	2353	2350	2310 / 2	32	9	12
C7552	2118	1968	1863 / 4	5583	20	18

mcnc.genlib. The results of verifying these transformed circuits are obtained by running our program on a Sun Sparc-20 workstation equipped with 128MB memory. It can be seen that every circuit can be verified within one minute of CPU time (which includes the random simulation time), indicating that our approach is quite robust and applicable to circuits that have gone through intensive logic transformations. Our experience shows that it is very common that some equivalent signal pairs cannot be identified unless we backward expand the dynamic support for a number of levels. In the worst case, our approach will degenerate to the global BDD-based approach. However, the memory explosion problem does not occur for the entire set of ISCAS-85 benchmark circuits because many internal equivalent pairs can be efficiently identified. Column 4 and column 5 list the number of candidate key signal pairs (i.e., signal pairs with the same name) and the total number of equivalent and complementary signal pairs identified. It can be seen that the ratio of key signal pairs to the total number of equivalent pairs is very small, indicating that our program does not rely on the naming information very much in these cases.

Table 2 shows the results of using AQUILA to verify the circuits after redundancy removal in SIS. It can be seen that AQUILA are much faster than the one presented in [34]. Also, the results are quite comparable to the ones presented in [30], which was developed independently around the same time as this work and run on a Sun-workstation.

Table 3 shows the results of using our program to verify ISCAS89 benchmark sequential circuits optimized by *script.rugged* in SIS. Note that, even though *script.rugged* is mostly combinational optimization, it could change the number of flip-flops. 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 applied, even if the inputs (outputs) of the flip-flops are treated as pseudooutputs (pseudoinputs). Our program automatically verified that these optimized circuits are indeed equivalent to their original version directly.

Several key columns of Table 3 are explained as follows:

1. *nodes (original/optimized):* are the numbers of nodes in the original and the optimized circuits, respectively. The optimized circuits are cleaned up by SIS

TABLE 3
Results of Verifying Sequential Circuits Optimized by *script.rugged* in SIS

Circuit	# nodes original / optimized	# FFs orig. / opt.	# internal equiv. pairs (comb / seq)	# equivalent FF-pairs (comb. / seq.)	# equivalent PO-pairs (comb. / seq.)	Simulation Time (seconds)	Verification Time (seconds)
s208	66 / 42	8 / 8	9 / 0	8 (8 / 0)	1 (1 / 0)	1	1
s298	75 / 65	14 / 14	17 / 0	14(14/ 0)	6 (6 / 0)	1	2
s344	114 / 102	15 / 15	29 / 0	15 (15 / 0)	11(11 / 0)	1	1
s349	117 / 101	15 / 15	29 / 0	15 (15 / 0)	11 (11 / 0)	1	1
s382	101 / 92	21 / 21	28 / 0	21 (21 / 0)	6 (6 / 0)	1	1
s386	118 / 60	6 / 6	7 / 0	6 (6 / 0)	7 (7 / 0)	1	1
s400	108 / 88	21 / 21	27 / 0	21 (21 / 0)	6 (6 / 0)	1	1
s420	140 / 84	16 / 16	19 / 0	16 (16 / 0)	1 (1 / 0)	1	1
s444	121 / 85	21 / 21	25 / 0	21 (21 / 0)	6 (6 / 0)	1	1
s510	179 / 115	6 / 6	8 / 0	6 (6 / 0)	7 (7 / 0)	1	1
s526	141 / 106	21 / 21	31 / 0	21 (21 / 0)	6 (6 / 0)	1	1
*s641	128 / 100	**19 / 17**	4 / 0	17 (2 / 15)	23 (16 / 7)	1	11
*s713	154 / 99	**19 / 17**	4 / 0	17 (2 / 15)	23 (14 / 9)	1	12
s820	256 / 137	5 / 5	16 / 0	5 (5 / 0)	19 (19 / 0)	1	1
s832	262 / 130	5 / 5	16 / 0	5 (5 / 0)	19 (19 / 0)	1	1
s838	288 / 162	32 / 32	39 / 0	32 (32 / 0)	1 (1 / 0)	1	1
s1196	389 / 273	18 / 18	40 / 0	17 (17 / 0)	14 (14 / 0)	2	3
s1238	429 / 286	18 / 18	46 / 0	18 (18 / 0)	14 (13 / 1)	2	4
s1423	491 / 390	74 / 74	109 / 0	72 (74 / 0)	5 (5 / 0)	16	3
s1488	550 / 309	6 / 6	29 / 0	6 (6 / 0)	19 (19 / 0)	2	3
s1494	558 / 305	6 / 6	30 / 0	6 (6 / 0)	19 (19 / 0)	2	3
*s5378	1074 / 858	**164 / 162**	223 / 0	162 (142 / 20)	49 (49 / 0)	14	12
s9234	1081 / 599	135 / 135	156 / 0	135 (135 / 0)	39 (39 / 0)	7	5
*s13207	2480 / 1175	**490 / 453**	301 / 32	419 (379 / 40)	121 (77 / 44)	284	122
*s15850	3379 / 2435	**563 / 540**	608 / 20	526 (523 / 3)	87 (73 / 14)	545	634
s35932	12492 / 7050	1728 / 1728	2048 / 0	1728 (1728 / 0)	320 (320 / 0)	386	75
s38417	8623 / 7964	1464 / 1464	1860 / 0	1464 (1464 / 0)	106 (106 / 0)	284	241

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

command "sweep" and then decomposed into AND/OR gates.

2. # *FFs (orig./opt.):* are 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 those sequentially equivalent pairs play an important role in reducing the verification 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 CPU times in seconds are given in the last two columns.

6.2 Verify Sequential Transformations

Table 4 shows the results of verifying ISCAS-89 benchmark circuits optimized using the information of unreachable states extracted using command "*extract_seq_dc*" in SIS. Our program successfully verified every circuit that can be sequentially optimized (i.e., circuit for which the unreachable states can be extracted by symbolic reachability analysis). Only four out of 19 optimized circuits are still combinationally equivalent to their respective original version. The rest are sequentially equivalent. If we only check combinational equivalence, the inequivalent primary output pairs for these circuits can be easily identified in the functional simulation stage. We compare the CPU time with the verification program in SIS as indicated in the last two columns.

Table 5 shows the results of verifying circuits after *script.rugged-and-then-retiming* transformation in SIS. No reset state is assumed for these circuits and the definition of 3-valued equivalent [16] is used. For these cases, our program heavily relies on comparing the signal names to pair up candidate equivalent FF-pairs or internal signal pairs. The preprocessing algorithm described in [17] is applied first to reduce the complexity of verifying retimed circuits. We found a distinguishing sequence to disprove

TABLE 4
Reaults of Verifying ISCAS-89 Circuits Optimized with Sequential Don't Care Information in SIS

Circuit	# nodes original / optimized	# FFs orig. / opt.	#internal equiv. pairs (comb / seq)	# equivalent FF-pairs (comb. / seq.)	# equivalent PO-pairs (comb. / seq.)	combinational equivalence ?	AQUILA Time (sec)	SIS Time (sec)
s208	66 / 16	8 / 8	15 / 0	8 / 0	1 / 0	YES	1	3
s298	75 / 22	14 / 14	8 / 10	14 / 0	6 / 0	NO	1	2
s344	114 / 51	15 / 15	44 / 2	15 / 0	9 / 2	NO	1	5
s349	117 / 52	15 / 15	41 / 2	15 / 0	9 / 2	NO	1	5
s382	101 / 40	21 / 21	24 / 7	21 / 0	6 / 0	NO	2	47
s386	118 / 25	6 / 6	4 / 5	6 / 0	2 / 5	NO	1	1
s400	108 / 38	21 / 21	22 / 8	21 / 0	6 / 0	NO	2	48
s420	140 / 37	16 / 16	29 / 0	16 / 0	1 / 0	YES	41	1557
s444	121 / 36	21 / 21	22 / 6	21 / 0	6 / 0	NO	1	31
s510	179 / 31	6 / 6	21 / 2	6 / 0	6 / 1	NO	5	2
s526	141 / 40	21 / 21	7 / 23	21 / 0	6 / 0	NO	18	32
s641	128 / 58	19 / 17	64 / 1	17 / 0	22 / 1	NO	1	9
s713	154 / 57	19 / 17	70 / 1	17 / 0	22 / 1	NO	1	9
s820	256 / 56	5 / 5	25 / 7	5 / 0	15 / 4	NO	3	3
s832	262 / 54	5 / 5	29 / 8	5 / 0	14 / 5	NO	4	3
s1196	389 / 102	18 / 18	73 / 0	18 / 0	14 / 0	YES	3	8
s1238	429 / 108	18 / 18	77 / 0	18 / 0	14 / 0	YES	3	8
s1488	550 / 104	6 / 6	44 / 4	6 / 0	10 / 9	NO	4	5
s1494	558 / 107	6 / 6	45 / 5	6 / 0	11 / 8	NO	4	5

the equivalence of transformed circuit s5378 and its reference version by running *stg3* on the miter. The distinguishing sequence contains two input vectors. Note that the inequivalence could be due to a potential bug in the retiming program is SIS or may result from our format translator.

6.3 Verify a Gate-Level Implementation against Its Structural RTL Specification

Table 6 shows the result of verifying an industrial example from Rockwell. We compared the *combinational* function of a arithmetic gate level netlist (extracted from a highly customized transistor-level circuit) against a golden simple RTL description. In the first step, we synthesize the RTL specification into a gate level netlist using Synopsys Design Compiler. Then, we perform gate-to-gate equivalence check between the customized netlist and the gate level specification. Our program successfully concludes in 66 seconds that the customized implementation functionally conforms to its RTL specification, while the verification program in SIS fails.

To demonstrate its capability for handling partially different encoded circuits with an external don't care set, we implemented a 2's complement shift-and-add multiplier based on the functional description given in [14]. The entire flow of our experiment is shown in Fig. 21.

In the functional description, the data-path and the controller are separated. The data-path is given at the micro-architecture level consisting of a number of RTL components. The controller is described by a state transition table with a number of unspecified transitions. The external don't cares associated with the controller are extracted from the incompletely specified functional specification. Some of these don't cares are due to the assumption of environment and some are due to the interaction between the data-path and the controller. We then synthesized the controller using the state encoding program, *nova*, to generate the specification and don't care network. On the other hand, we use one-hot encoding to synthesize the controller of the implementation. As a consequence, the implementation has three more flip-flops than the specification. If correctly synthesized, the cycle behavior of two circuits for all care sequences are the same. Note that our approach is *not* intended to verify two designs with different cycle behaviors. The data-paths of the specification and the implementation are realized using the same type of RTL modules. We then further optimize the implementation using the optimization script *script.rugged* in SIS. Since the specification network and the implementation network interpret the external don't care conditions differently, they can be shown as inequivalent by simulation if don't cares are not taken into account during verification. AQUILA proves the optimized implementation is indeed equivalent to the specification in the care space.

The results are shown in Table 7. The first four columns show the circuit statistics. The last two columns show the verification times using the verification program in SIS and AQUILA. For SIS verification times, we report the times to verify two identical implementation circuits because SIS does not cope with the don't care issues. Due to the huge state space created by the data-path, pure BDD-based state enumeration techniques run into a memory explosion on a workstation with 128 Mbyte memory for the 16-bit and 32-bit shift-and-add multipliers. AQUILA completes the verification tasks in several seconds because it is not sensitive to the number of flip-flops in the data path, even though the equivalence checking is conducted at the gate-

TABLE 5
Results of Verifying Circuits *optimized-and-then-retimed* in SIS

Circuit	# connections original / optimized+retimed	# FFs original / optimized+retimed	3-valued equivalent (?)	CPU-Time (seconds)
s27	9 / 9	3 / 3	YES	1
s208*	66 / 16	8 / 9	YES	1
s298*	81 / 29	14 / 23	YES	3
s344*	114 / 55	15 / 27	YES	1
s349*	117 / 56	15 / 27	YES	1
s382*	105 / 45	21 / 30	YES	2
s386*	118 / 29	6 / 14	YES	2
s400*	112 / 43	21 / 23	YES	2
s420*	140 / 33	16 / 19	YES	65
s444*	125 / 42	21 / 25	YES	2
s510	179 / 31	6 / 6	YES	4
s526*	147 / 49	21 / 39	YES	45
s641	130 / 60	19 / 17	YES	2
s713	156 / 59	19 / 17	YES	2
s820	256 / 56	5 / 5	YES	3
s832	262 / 54	5 / 5	YES	3
s1196	390 / 103	18 / 18	YES	6
s1238	430 / 109	18 / 18	YES	6
s1488	550 / 105	6 / 6	YES	6
s1494	558 / 108	6 / 6	YES	6
s1423*	491 / 141	74 / 86	YES	8
s5378*	1074 / 392	164 / 343	NO	100.6 (by stg3)
s9234.1	1081 / 229	211 / 211	YES	29

**Circuits whose clock times have been improved by the retiming-program in SIS.*

TABLE 6
Result of RTL-to-Gate Verification for an Industrial Design

Circuit name	# inputs	# outputs	Specification # node	Implement. # nodes	# same-name pairs	# equivalent signal pairs (equiv. / complement.)	CPU-time (seconds)
R1	98	48	528	893	0	132 / 32	66

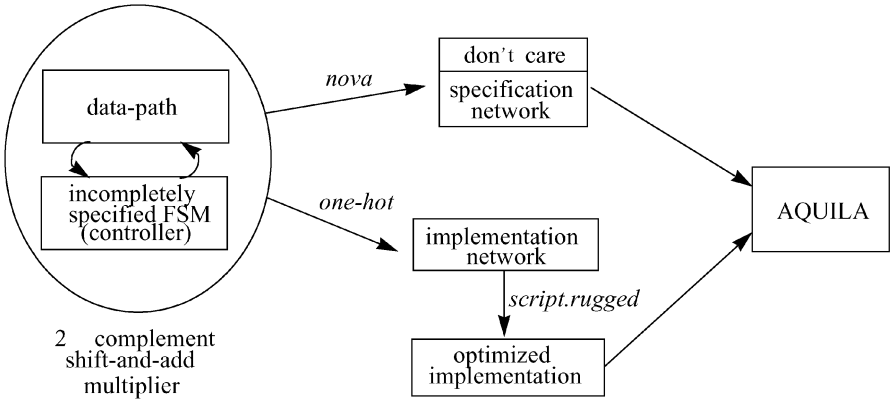


Fig. 21. The flow of verifying a 2's complement multiplier to its structural RTL specification.

TABLE 7
Results of Verifying Shift-and-Add Multipliers with External Don't Cares

# Bits	# FFs spec. / impl.	# Nodes spec / impl	# Literals spec. / impl.	SIS verification time (seconds)	Our verification time (seconds)
4-bit	19 / 22	147 / 119	469 / 293	*6.8	1.8
16-bits	57 / 60	421 / 346	1249 / 846	—	3.4
32-bits	105 / 108	745 / 613	2214 / 1483	—	7.0

** verification time of identical circuits in SIS.*

TABLE 8
Results of Verifying a Reencoded Simple Processor-Like Design

Data-width	#FFs spec. / impl.	#Nodes spec / impl	#Literals spec. / impl.	SIS verification time (seconds)	Our verification time (seconds)
8-bit	94 / 92	1977 / 536	4363 / 1240	1887	7.7
16-bits	174 / 174	3761 / 979	8027 / 2271	—	15.7
21-bits	224 / 224	4877 / 1259	10317 / 2916	—	38.3

–: cannot complete because of memory explosion on a 128 Mbyte workstation.

level. Increasing the data-width of the data-path does not increase the overall complexity of verification significantly.

Table 8 shows the experimental results of verifying a simple microprocessor design, which contains a register file, a decoder, a data-path, and a controller. The original specification is described in Verilog at the RT level. We reencode the controller using the same number of bits as a new implementation. We then synthesize both specification and implementation and output their gate-level networks in BLIF format using VIS [40]. Also, *script.rugged* is applied to optimize the implementation. Note that the flip-flops of a design will be removed when it is synthesized from Verilog to a gate-level implementation in BLIF format in VIS. However, these flip-flops can be recovered using some information generated by VIS. The verification times using SIS and our tool are shown in the last two columns. Since no external don't care is utilized for optimization, SIS can verify the example with 8-bit data width, but again will fail on larger ones. We do not report designs with even larger data-width because we have some difficulty in using the Verilog to BLIF translator in VIS when we try to further scale up the data-width beyond 21-bits.

7 Conclusion

Traditionally, either explicit or implicit product machine traversal are performed to verify the functional equivalence of two sequential designs. These approaches are subject to combinatorial explosion and, thus, are only suitable for small to medium-sized designs. In this paper, we present a unified approach that combines the advantages of FSM traversal and incremental verification techniques to enhance the capability of formal equivalence checking. Our approach can perform both gate-to-gate and RTL-to-gate verification. For gate-to-gate verification, we devise a local BDD-based engine to explore the combinational similarity, as well as the sequential similarity between designs. This technique is less sensitive to the degree of structural similarity as compared to the pure ATPG-based approaches. Therefore, it is more suitable for verifying designs that have gone through intensive logic optimization. For RTL-to-gate verification, we further developed techniques to address two issues: 1) external don't care modeling and utilization, and 2) a partial set of unreachable states extraction. Based on these two techniques, we are able to extend our gate-to-gate equivalence checker to RTL-to-gate verification. Experimental results show that this approach has great potential to handle a lot of practical designs beyond the capability of existing methods.

Acknowledgments

This work was supported in part by the US National Science Foundation under Grant MIP-9503651 and in part by the California MICRO program through Fujitsu/Rockwell. S.Y. Huang was with the University of California, Santa Barbara, when this work was performed.

References

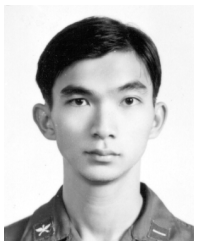
[1] M. Abramovici, M.A. Breuer, and A.D. Friedman, *Digital Systems Testing and Testable Design.* IEEE Press, 1990.
[2] P. Ashar, A. Gupta, and S. Malik, "Using Complete-1-Distinguishability for FSM Equivalence Checking," *Proc. Int'l Conf. Computer-Aided Design,* pp. 346-353, Nov. 1996.
[3] R.A. Bergamaschi, D. Brand, L. Stok, M. Berkelaar, and S. Prakash, "Efficient Use of Large Don't Cares in High-Level and Logic Synthesis," *Proc. Int'l Conf. Computer-Aided Design,* pp. 272-278, Nov. 1995.
[4] C.L. Berman and L.H. Trevillyan, "Functional Comparison of Logic Designs for VLSI Circuits," *Proc. Int'l Conf. Computer-Aided Design,* pp. 456-459, Nov. 1989.
[5] D. Brand, "Verify Large Synthesized Designs," *Proc. Int'l Conf. Computer-Aided Design,* pp. 534-537, Nov. 1993.
[6] R.E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," *IEEE Trans. Computers,* vol. 35, no. 8, pp. 677-691, Aug. 1986.
[7] K.-T. Cheng, "Redundancy Removal for Sequential Circuits without Reset States," *IEEE Trans. Computer-Aided Design,* pp. 652-667, Jan. 1993.

[8] W.-T. Cheng, "The BACK Algorithm for Sequential Test Generation," *Proc. Int'l Conf. Computer Design,* pp. 66-69, Oct. 1988.
[9] H. Cho, G.D. Hachtel, S.W. Jeong, B. Plessier, E. Schwarz, and F. Somenzi, "ATPG Aspects of FSM Verification," *Proc. Int'l Conf. Computer Design,* pp. 134-137, Nov. 1990.
[10] H. Cho and F. Somenzi, "Sequential Logic Optimization Based on State Space Decomposition," *Proc. European Conf. Design Automation,* pp. 200-204, Feb. 1993.
[11] O. Coudert, C. Berthet, and J.C. Madre, "Verification of Synchronous Sequential Machines Based on Symbolic Execution," *Proc. Automatic Verification Methods for Finite State System,* 1990.
[12] C.A. Eijk and J.A. Jess, "Detection of Equivalent State Variables in Finite State Machine Verification," *Notes Int'l Workshop Logic Synthesis,* pp. 3.35-3.44, 1995.
[13] A. Ghosh, S. Devadas, and A.R. Newton, "Test Generation and Verification for Highly Sequential Circuits," *IEEE Trans. Computed-Aided Design,* pp. 652-667, May 1991.
[14] J.P. Hayes, *Computer Architecture and Organization,* second ed. McGraw-Hill, 1988.
[15] Y.V. Hoskote, "Formal Techniques for Verification of Synchronous Sequential Circuits," PhD thesis, Univ. of Texas at Austin, Dec. 1995.
[16] S.-Y. Huang, K.-T. Cheng, K.-C. Chen, and U. Glaeser, "An ATPG-Based Framework for Verifying Sequential Equivalence," *Proc. Int'l Test Conf.,* pp. 865-874, Oct. 1996.
[17] S.-Y. Huang, K.-T. Cheng, and K.-C. Chen, "On Verifying the Correctness of Retimed Circuits," *Proc. Great-Lake Symp. VLSI,* pp. 277-281, Mar. 1996.
[18] S.-Y. Huang, K.-C. Chen, and K.-T. Cheng, "Error Correction Based on Verification Techniques," *Proc. Design Automation Conf.,* pp. 258-261, June 1996.
[19] S.-Y. Huang, K.-T. Cheng, and K.-C. Chen, "AQUILA: An Equivalence Verifier for Large Sequential Circuits," *Proc. Asia and South Pacific Design Automation Conf.,* pp. 455-460, Jan. 1997.
[20] J. Jain, R. Mukherjee, and M. Fujita, "Advanced Verification Techniques Based on Learning," *Proc. Design Automation Conf.,* pp. 420-426, June 1995.
[21] A. Kuehlmann, A. Srinivasan, and D.P. LaPotin, "Verity—A Formal Verification Program for Custom CMOS Circuits," *IBM J. Research and Development,* vol. 39, pp. 149-165, Jan./Mar. 1995.
[22] W. Kunz, "HANNIBAL: An Efficient Tool for Logic Verification Based on Recursive Learning," *Proc. Int'l Conf. Computer-Aided Design,* pp. 538-543, Nov. 1993.
[23] C.E. Leiserson and J.B. Laxe, "Retiming Synchronous Circuitry," *Algorithmica,* vol. 6, no. 1, 1991.
[24] R. Marlett, "EBT: A Comprehensive Test Generation Technique for Highly Sequential Circuits," *Proc. Design Automation Conf.,* pp. 332-339, June 1978.
[25] J. Jain, R. Mukherjee, and M. Fujita, "Advanced Verification Techniques Based on Learning," *Proc. Design Automation Conf.,* pp. 420-426, June 1995.
[26] A. Kuehlmann, A. Srinivasan, and D.P. LaPotin, "Verity—A Formal Verification Program for Custom CMOS Circuits," *IBM J. Research and Development,* vol. 39, pp. 149-165, Jan./Mar. 1995.
[27] W. Kunz, "HANNIBAL: An Efficient Tool for Logic Verification Based on Recursive Learning," *Proc. Int'l Conf. Computer-Aided Design,* pp. 538-543, Nov. 1993.
[28] C.E. Leiserson and J.B. Laxe, "Retiming Synchronous Circuitry," *Algorithmica,* vol. 6, no. 1, 1991.
[29] R. Marlett, "EBT: A Comprehensive Test Generation Technique for Highly Sequential Circuits," *Proc. Design Automation Conf.,* pp. 332-339, June 1978.
[30] Y. Matsunaga, "An Efficient Equivalence Checker for Combinational Circuits," *Proc. Design Automation Conf.,* pp. 629-634, June 1996.
[31] C. Pixley, "A Theory and Implementation of Sequential Hardware Equivalence," *IEEE Trans. Computer-Aided Design,* pp. 1,469-1,494, Dec. 1992.
[32] C. Pixley, V. Singhal, A. Aziz, and R.K. Brayton, "Multi-Level Synthesis for Safe Replaceability," *Proc. Int'l Conf. Computer-Aided Design,* pp. 442-449, Nov. 1994.
[33] D.K. Pradhan, D. Paul, and M. Chatterjee, "VERILAT: Verification Using Logic Augmentation and Transformations," *Proc. Int'l Conf. Computer-Aided Design,* pp. 88-95, Nov. 1996.
[34] S.M. Reddy, W. Kunz, and D.K. Pradhan, "Novel Verification Framework Combining Structural and OBDD Methods in a Synthesis Environment," *Proc. Design Automation Conf.,* pp. 414-419, June 1995.
[35] R. Rudell, "Dynamic Variable Ordering for Ordered Binary Decision Diagram," *Proc. Int'l Conf. Computer-Aided Design,* pp. 42-47, Nov. 1993.
[36] N. Shenoy and R. Rudell, "Efficient Implementation of Retiming," *Proc. Int'l Conf. Computer-Aided Design,* pp. 226-233, Nov. 1994.
[37] V. Singhal and C. Pixley, "The Verification Problem for Safe Replaceability," *Proc. Conf. Computer-Aided Verification,* pp. 311-323, June 1994.
[38] "SIS: A System for Sequential Circuit Synthesis,"Report M92/41, Univ. of California, Berkeley, May 1992.
[39] H.J. Touati, H. Savoj, B. Lin, R.K. Brayton, and A. Sangiovanni-Vincentelli, "Implicit State Enumeration of Finite State Machines Using BDD's," *Proc. Int'l Conf. Computer-Aided Design,* pp. 130-133, Nov. 1990.
[40] R.K. Brayton et al., "VIS: Verification Interacting with Synthesis," *Proc. Conf. Computer-Aided Verification,* pp. 408-412, 1996.

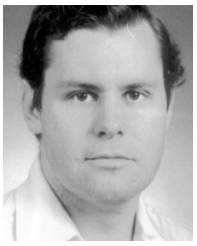
Shi-Yu Huang received the BS degree in Electrical Engineering from National Taiwan University in 1988, and the MS and PhD degrees in electrical and computer engineering from the University of California, Santa Barbara, in 1992 and 1997, respectively. From 1997 to 1998, he was a software engineer at National Semiconductor Corp., investigating the system-on-a-chip design methodology. He coauthored a book entitled *Formal Equivalence Checking and Design Debugging*. In 1999, he joined the National Tsing-Hua University, Taiwan, Republic of China, as an assistant professor. His research interests are in the area of computer-aided design for VLSI with an emphasis on formal verification.

Kwang-Ting (Tim) Cheng received the BS degree in electrical engineering from National Taiwan University in 1983 and the PhD degree in electrical engineering and computer science from the University of California, Berkeley, in 1988. He worked at AT&T Bell Laboratories in Murray Hill, New Jersey, from 1988 to 1993. He joined the faculty at the University of California, Santa Barbara, in 1993, where he is currently a professor of electrical and computer engineering. His current research interests include VLSI testing, design synthesis, and design verification. He has published more than 120 technical papers, coauthored three books and holds six US patents in these areas. He has also been working closely with US industry for projects in these areas. He received the Best Paper award at the 1994 Design Automation Conference and the Best Paper award at the 1987 AT&T Conference on Electronic Testing. He currently serves on the Editorial Boards of the *IEEE Transactions on Computer-Aided Design*, *IEEE Design and Test of Computers*, and the *Journal of Electronic Testing: Theory and Applications*. He has been general chair and program chair of the IEEE International Test Synthesis Workshop. He has also served on the technical program committees for several international conferences on CAD and testing.

Kuang-Chien Chen received the BS degree in electrical engineering from National Taiwan University, Taiwan, in 1983, and the MS and PhD degrees in computer science from the University of Illinois at Urbana-Champaign in 1987 and 1990, respectively. He is currently with Verplex Systems, Inc., Santa Clara, California, where he is engaged in the research and development of advanced verification tools. Prior to joining Verplex, he held research and management positions at Fujitsu Laboratories of America, Inc., and had been working in the area of formal verification, synthesis, and logic optimization. His main interest now is building automatic verification tools based on equivalence checking, model checking, BDD, and ATPG-related techniques. Dr. Chen is a member of the ACM.



Chung-Yang Huang received the BS degree in electrical engineering from National Taiwan University, Taiwan, in 1992. He joined the Electrical and Computer Engineering Department of the University of California, Santa Barbara, in 1995 and is now a PhD candidate. Between 1996 and 1997, he worked with Avant! Corp. for post-layout timing optimization using logic restructuring techniques. Since then, he has been working with Verplex Systems, Inc. in developing a model checking tool. His research interests include RTL design verification, automatic test pattern generation, and post-layout performance optimization.



Forrest Brewer received the BS degree in physics with honors from the California Institute of Technology in 1980, and the MS and PhD degrees in computer science in 1985 and 1988, respectively, from the University of Illinois at Urbana-Champaign. His current research work is the application of Boolean symbolic techniques in high-level synthesis. His work includes synthesis of production based specifications, exact models of control-dependent scheduling, systematic analysis of speculative execution, and data-path constrained synthesis. Recent work includes nondeterministic automata scheduling analysis for complex digital systems. Applications of this work range from microcontroller synthesis to the underlying technology of the Synopsys Protocol Compiler tool. Professor Brewer is a member of the IEEE, the ACM, the American Physical Society, and Tau Beta Pi. He is currently an associate professor in the Department of Electrical and Computer Engineering at the University of Caligornia, Santa Barbara.