Formal Verification of Combinational Circuits

Jawahar Jain[1] Amit Narayan[2] M. Fujita[1] A. Sangiovanni-Vincentelli[2]

With the increase in the complexity of present day systems, proving the correctness of a design has become a major concern. Simulation based methodologies are generally inadequate to validate the correctness of a design with a reasonable confidence. More and more designers are moving towards formal methods to guarantee the correctness of their designs. In this paper we survey some state-of-the-art techniques used to perform automatic verification of combinational circuits.

We classify the current approaches for combinational verification into two categories: functional and structural. The functional methods consist of representing a circuit as a canonical decision diagram. Two circuits are equivalent if and only if their decision diagrams are equal. The structural methods consist of identifying related nodes in the circuit and using them to simplify the problem of verification. We briefly describe some of the methods in both the categories and discuss their merits and drawbacks.

1 Introduction

Successful design of a complex digital system requires verifying the correctness of the implementation with respect to its intended functionality. Traditionally, the task of design validation is carried out by means of simulation. In a simulation based approach, the designer needs to create a complete set of test vectors which represents all possible inputs to the system. The outputs for each of these test vectors are analyzed to guarantee the correctness of the design. This process is highly CPU-time intensive; in almost all practical situations it is infeasible to exhaustively simulate a design to guarantee its correctness.

Due to the limitations of a simulation based approach, various formal verification strategies are becoming increasingly popular. By using these techniques, it is possible to guarantee the correctness of a design under all possible input combinations.

The process of designing a complex system usually starts with an abstract model of the system. This model is subjected to extensive simulation after which it becomes the "golden specification" of the design. From this abstract model, a detailed implementation is derived in a hierarchical manner. First the abstract model is translated into a synthesizable behavioral RTL model representing the block structure behavior of the design. This behavioral RTL model is then translated into a structural model which is a logic level description of the system. From the structural RTL model a transistor netlist and subsequently the physical layout of the design is derived.

In a successful design methodology it is essential to catch bugs early in the design cycle. For this, the functionality of the design is verified at every level of hierarchy against the original ("golden") specification. This kind of formal verification in which different implementations of the same design are compared to check their equivalence is known as implementation verification. Implementation verification typically proceeds in two phases. In the first phase, a Boolean network representing the original design is extracted from the RTL description or the transistor level netlist [35, 13, 14, 33, 51]. In the second phase, the correctness of this Boolean network is verified using some formal methods.

In this paper we will focus only on the second phase. We will describe some recent advances made in the area of verifying the equivalence of two Boolean networks. More specifically, we will focus only on the verification of combinational circuits i.e., circuits in which the outputs depend only on the current inputs (as opposed to sequential circuits in which the outputs depend not only on the present inputs but also on the past sequence of inputs). Some sequential verification problems can also be reduced to a combinational verification problem (e.g. when the corresponding latches in the two designs can be identified). Although techniques exist for verifying general sequential circuits, currently it is not practical to verify large industrial designs using them.

The combinational verification problem can be stated as follows: Given two Boolean netlists, check if the corresponding outputs of the two circuits are equal for all possible inputs. This problem is NP-hard and hence a general solution which can handle arbitrary Boolean functions is not likely to exist. However, since the functions that are implemented in practice are not random Boolean functions, various techniques have been developed which can successfully verify large designs.

The work in equivalence checking can be classified into two main categories:

- The first approach consists of transforming the output functions of the two networks into a unique (i.e. canonical) representation. Two circuits are equivalent if and only if the canonical representations of the corresponding outputs are the same. The most popular canonical representations are based on Binary Decision Diagrams (BDDs). We will discuss methods based on BDDs in Section 2. In the worst case these methods can require exponential space (in the number of inputs). We will discuss some techniques for dealing with this "memory explosion" problem in BDD representations.
- The second approach consists of identifying equivalent points and implications between the two circuits. Using this information the process of equivalence checking can be simplified. Since a typical design proceeds by a series of local changes, in most cases there are a large number of implications between the two circuits to be verified. These implication based techniques have been very successful in verifying large circuits and form the basis of most combinational verification systems. We will discuss some of these techniques in section 3.

2 Methods Based on Decision Diagrams

In this approach, the output functions of the two networks are represented as canonical BDDs. The two circuits are equivalent if and only if the BDDs of their corresponding outputs are equal (i.e. isomorphic).

A BDD over a set of $X_n = \{x_1, \ldots x_n\}$ of Boolean variables is a directed acyclic graph with one source and at most two sinks

[1]Fujitsu Laboratories of America, Santa Clara, CA 95054

[2]Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, CA 94720

labeled by 0 and 1. Each non-sink (internal) node is labeled by a variable in X_n and has two outgoing edges - corresponding to where the variable evaluates to a 0 or to a 1. For a given assignment to the variables, the function value is evaluated by tracing a path from the root to the terminal. For a given input $m = (m_1, \ldots, m_n)$, the evaluation starts at the root and at an internal node with label x_i the outgoing edge with label m_i is chosen (see Figure 1).

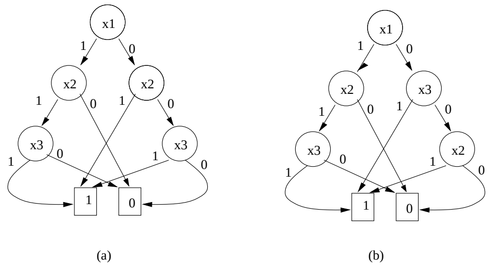

Figure 1: (a) ROBDD and (b) Free BDD

Though BDDs have been researched for about four decades [39, 1], they found widespread use only after Bryant [12] showed that such graphs, under two restrictions, are canonical and can be easily manipulated. The first restriction is that a total ordering of the variables is enforced in the graph. That is, if we consider variables to be ordered as $x_1 < x_2 < \ldots < x_n$, then every path from the root to a sink encounters the variables in that order. The second restriction is that the graph is reduced. A graph can be reduced by the repeated application of the following two rules until they are no longer applicable. These rules are:

- Merging Rule: Two isomorphic subgraphs should be merged.
- Deletion Rule: A vertex whose two branches point to the same vertex should be deleted.

The resulting BDD is called a Reduced Ordered BDD (an ROBDD). The important symbolic manipulation procedures introduced by Bryant were *apply* and *compose*; these techniques operate on two identically ordered ROBDDs. *Apply* allows two ROBDDs to be combined under some Boolean operation, and *compose* allows the substitution of an ROBDD variable with a function.

Since ROBDDs are canonical, they can be used directly for checking the equivalence of two Boolean circuits. Two circuits are equivalent if and only if the ROBDDs representing the corresponding outputs of the two circuits are equal. ROBDDs are typically constructed using some variant of Bryant's *apply* procedure [12]; the ROBDD for a gate g is synthesized by the symbolic manipulation of the ROBDDs of its inputs, based on the functionality of g. The gates of the circuit are processed in a depth-first manner until the ROBDDs of the desired output gate(s) are constructed. For details on ROBDDs, and the implementation of a typical ROBDD package, please refer to [9, 12, 15].

Although ROBDDs are canonical and hence can directly be used for combinational verification, their construction is often a time and memory intensive process. The size of an ROBDD representing a Boolean function can be exponential in the number of primary inputs in the worst case. This problem is commonly referred to as the "memory explosion" problem. In the following sections we will discuss various methods which deal with the memory explosion problem during ROBDD construction.

2.1 Variable Ordering

The size of an ROBDD is strongly dependent on the ordering of its variables. Much of the prior research in ROBDDs has focused on finding good variable orders to reduce the size of an ROBDD representing a Boolean function. Given a combinational netlist, [21, 23] discuss some heuristics for ordering the primary input variables which lead to a compact ROBDD representation of the outputs. These techniques for the first time successfully demonstrated that ROBDDs could be used for verifying large circuits. Another significant advance in variable ordering was made with the introduction of dynamic variable reordering [49]. In this procedure a periodic reordering of variables is attempted to reduce the memory requirement. Given a graph G, a variable v is successively moved to each position in the ordering list and the resulting graph size is examined. The variable is finally assigned the position which results in the smallest graph size. This process is known as *sifting* and is repeated for each variable in the graph. Sifting n variables, in a graph of size G, requires $O(n \cdot |G|)$ effort. A less expensive procedure may also be used where variables are reordered in a window of say, 3 consecutive variables. This window is then moved forward to include the next variable in the graph, and the process is repeated till all n variables have been considered. Improvements to sifting based reordering techniques were suggested by [47] where the number of sift operations were reduced by grouping, and thereby sifting together, the symmetric variable pairs. Further improvements were suggested in [46] where the concept of extended symmetry was introduced to group a larger block of variables.

Though computationally somewhat expensive, dynamic reordering techniques are widely used as they allow the variable ordering to adapt to the changing functions that are being represented. Although good variable ordering methods have considerably increased the class of functions which can be efficiently verified using ROBDDs, there are still many functions for which any ROBDD is exponential in the number of inputs (e.g. integer multiplier). Further, the problem of finding optimum variable orders is an intractable problem and there are many instances where although a good orderings might exist, the heuristics are unable to find them.

2.2 Breadth First Manipulation

In [45, 2], it was shown that by manipulating ROBDDs in a breadth-first fashion much larger ROBDDs can be processed than is possible by the conventional *apply* procedure [12] which operates as a depth first algorithm. This gain in memory is achieved by keeping only a few levels of ROBDDs in the main memory at any given time and storing the rest in the secondary memory which is typically much larger. Breadth first algorithms allow an orderly memory access. This results in fewer page faults and consequently a significant improvement in performance, especially for large circuits. Recently a complete package was implemented [50] using the breadth-first manipulation idea which gives an order of magnitude performance gain over the conventional ROBDD packages.

The main drawback of this approach is that as only a few levels are kept in the main memory at a time, it is difficult to dynamically reorder the ROBDD during an operation.

2.3 Node Decompositions

ROBDDs employ a decomposition known as the "Shannon Decomposition" in which a function f is decomposed in terms of a variable x as follows:

$$f = \overline{x} f_{\overline{x}} + x f_x \tag{1}$$

Here f_x represents the positive cofactor of f with respect to x and is obtaining by replacing variable x by the value 1. Similarly, $f_{\overline{x}}$ represents the negative cofactor with respect to x and is obtained by replacing x by 0.

Canonical but fundamentally different data structures such as ordered Functional Decision Diagrams (OFDDs) [34] and Ordered Kronecker Functional Decisions Diagrams (OKFDDs) [20] have also been proposed to extend the set of functions that can be efficiently symbolically manipulated. In OFDDs the function is decomposed using the "Reed-Muller" ("Davio") expansion. In this decomposition, the function f is represented as either:

$$f = f_{\overline{x}} \oplus (x(f_x \oplus f_{\overline{x}})); \quad or \quad f = f_x \oplus (\overline{x}(f_x \oplus f_{\overline{x}})) \qquad (2)$$

OFDDs are canonical like ROBDDs and hence can be used in verification. There are some functions for which ROBDDs are exponential but OFDDs are polynomial. Thus OFDDs extend the class of functions which can be verified in polynomial memory resources, but conversely there are functions for which OFDDs are exponentially larger than ROBDDs. OKFDDs try to benefit from both decompositions; each variable has an associated decomposition which can be either Reed-Muller or Shannon. Variables are ordered and every occurrence of a given variable must use the same decomposition. Although in theory OKFDDs can be exponentially more compact than both OFDDs and ROBDDs, in practice they seem to have provided only a modest improvement over ROBDDs (approx. 35%).

2.4 Non-Canonical BDDs

Non-canonical BDD representations such as XBDDs [32], gBDDs [3], IBDDs [27] have been explored to obviate OBDD memory explosion, often leading to more efficient verification. For example, in IBDDs any variable can appear multiple times on any path from the root to the terminal; an order is imposed on the multiplicity of the occurrence. In other words, an IBDD can be considered as "layered" BDD such that within each layer the appearances of variables obey a linear order as in ROBDDs. It was shown in [27] that some functions intractable for ROBDDs such as *hidden-weighted-bit* function, Booth-encoded as well as integer multiplier, etc. can be verified in polynomial time using IBDDs. However, due to lack of detailed experimental results and/or publicly available function manipulation packages, we feel that further research is warranted to gauge the true potential of these novel non-canonical data structures.

Another strategy to reduce the BDD sizes in function representation is to relax the total ordering requirement of ROBDDs. One such relaxation is to allow variables to occur in any order but at most once along any path from the root to the terminal. Such BDDs are called Free BDDs (Figure 1(b)). In general free BDDs are not canonical and their manipulation is an intractable problem [22]. However, in [25] it was shown that restricted forms of free BDDs known as typed-Free BDDs are canonical and can be easily manipulated. In typed-Free BDDs, for any given variable assignment, the resulting paths in all graphs contain variables in the same order. The variable ordering for different assignments might be different. Unfortunately, the practical problems in choosing a good *type* can greatly reduce the flexibility gained from relaxing the variable ordering constraints. Some heuristics for generating typed Free BDDs were presented in [7, 6]. Typed-free BDDs extend the class of functions which can be represented in polynomial space but there are still some practical functions for which Free BDDs are exponential (e.g. integer multiplier).

2.5 Partitioned ROBDDs

All the BDD methods discussed so far represent a function over the entire Boolean space as a single graph (rooted at a unique source). It was shown in [28, 44] that exponentially more compact representations can be obtained by partitioning the Boolean space and representing the functionality over each partition as a separate graph. This compactness in representation is achieved without sacrificing the desirable properties of the underlying graph which is used to represent each partition. In [44] this notion of partitioning was used to develop a data structure called partitioned-ROBDD in which every partition of the Boolean space is represented as an ROBDD. Different partitions can have different ordering. It was shown that partitioned-ROBDDs provide a compact, canonical and efficiently manipulable representation for Boolean functions. The notion of partitioning is general and can be applied to any BDD representation.

It was shown in [28, 44] that the class of functions representable in polynomial space by monolithic ROBDDs is strictly contained in the class of functions that have a polynomially sized partitioned-ROBDD representation. Similarly, the class of functions with polynomially sized Free BDDs is strictly contained in the class of functions with polynomially sized partitioned-Free BDDs. Note, partitioned-ROBDDs can be exponentially smaller than even free BDDs. Further, for combinational verification only one partition needs to be present in the memory at a given time. This further reduces the total memory requirement of verification. Using this representation, some industrial circuits could be verified for the first time [44]. Since this is a relatively new data structure, its full impact can be judged only with time.

2.6 Combining Bottom-up and Top-down approaches of ROBDD construction

Regardless of the choice of the BDD representation for verification, a critical issue that needs to be addressed is that of constructing the final canonical graph from a given Boolean netlist in the most memory efficient manner. In this section we discuss a mixed bottom-up/top-down approach for ROBDD construction which attempts to minimize the intermediate peak memory requirement during ROBDD construction. Though the following discussion is with respect to ROBDDs, it should be equally applicable to other BDD methods as well.

Traditionally, ROBDDs for a given netlist are built in a bottom-up manner. To construct the ROBDD for a given node, ROBDDs of all the nodes that are present in the transitive fan-in of that node are constructed in terms of the primary inputs before the ROBDD of the target node is constructed. In this method, the peak intermediate memory requirement often far exceeds the final (canonical) representation size of the given function. Although in combinational verification we are interested in obtaining the ROBDD of only the output, the large intermediate peak memory requirement often limits our ability to construct it. This places a limit on the complexity of circuits that can be verified using ROBDDs, and also usually dictates the time required for ROBDD construction. In [43, 31], techniques to reduce the intermediate peak memory requirement by a suitable combination of bottom-up and top-down approaches were presented. It was shown that by using these techniques, ROBDDs for many circuit outputs can be constructed for which the conventional method fails. In addition, it was observed that the reduction in peak memory is often accompanied by a significant speed up in the ROBDD construction process.

Let us look at an example where the memory requirement for a bottom-up scheme is exponential while the decomposition/composition approach requires only polynomial resources. Consider the function shown in Figure 2. Here f and g are two internal nodes. Assume that the ROBDD of g is exponential in terms of the primary inputs (PIs) for any variable ordering. Further assume that all the other internal nodes of the function require only polynomial memory resources. If we try to build the ROBDD of the primary output y in a bottom-up fashion, we will need to build the ROBDD of g in terms of the PIs. But since the ROBDD of g is exponential for any given variable ordering, the peak memory required in the bottom up scheme will be exponential. The functionality of y can be expressed in terms of f and g by the equation $y = f \vee (f \wedge g)$. This simplifies to $y = f$. Therefore, to construct the ROBDD of y we do not need to construct the ROBDD of g. We can introduce a new variable representing g and build the ROBDD of y in terms of this variable. Since g is eventually not present in y, we need not ever construct the ROBDD of g. This can let us get an exponential reduction in the peak memory requirement and extend the class of circuits that can be efficiently processed using ROBDDs.

The previous example shows that in a typical ROBDD construction procedure there is frequent functional simplification due to Boolean Absorption: $x \vee (x \wedge y) = x$ and Boolean Cancelation: $x \wedge \overline{x} \wedge y = 0$. This means that the final output function is often much simpler (i.e. has a smaller ROBDD) than the intermediate functions that are used to implement it. In fact it can be proved that for general Boolean functions, ROBDDs are almost always exponential. It is only a statistical coincidence that ROBDDs for the functions that are generally encountered in real life do not exhibit the worst case behavior. But there do exist useful circuits (like Multiplier) which exhibit the exponential worst case. The traditional bottom-up method will fail when any node in the transitive fan-in of the target node exhibits the worst case behavior.

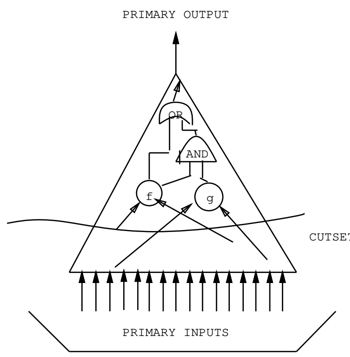

Figure 2: *Example where decomposition can avoid exponential blowup*

The procedures of [31] and [43] try to avoid building ROBDDs of the intermediate functions having a large ROBDD representation by introducing suitable decomposition points. The ROBDD of the output is then built in terms of these decomposition points and PIs. The functionality of the decomposition points is expressed as ROBDDs in terms of previously introduced decomposition points and PIs. Finally, the decomposition points are composed back to obtain a canonical ROBDD of the output function.

Two issues need to be addressed here:

- Finding a good decomposition set
- Determining a good order of composition of the decomposition variables to get the monolithic representation so that the intermediate memory explosion during the composition phase is low.

Heuristics for introducing good structural and functional decomposition points were described in [31] and for finding good order of composition were discussed in [43].

A reduction in memory is achieved since the intermediate points of large ROBDD sizes are avoided and also because dynamic variable reordering has to focus only on the target function and hence is more effective. This approach is fully compatible with other approaches of reducing memory (like variable ordering) and can be seamlessly integrated within any ROBDD package. Therefore, there seems to be no apparent trade-off in using it.

2.7 Probabilistic Verification

Another important way of verifying two circuits is to probabilistically check their equivalence [8, 29]. In probabilistic verification, every minterm of a function f is converted into an integer value under some random integer assignment ρ to the input variables. All the integer values are then arithmetically added to get the hash code $H_\rho(f)$ for f. One can assert, with a low probability of error, that $f \equiv g$ *iff* $H_\rho(f) = H_\rho(g)$. [8] suggested probabilistic verification of Boolean functions through hashing their free BDD representation to an integer value under some random integer assignment ρ to the input variables. Alternately, it was shown in [29] that we can also transform arbitrary representation of Boolean functions by first interpreting the given function as an integer-valued arithmetic expression. This arithmetic expression can then be evaluated on integer assignments to its input variables. By using the properties of such integer-valued arithmetic transformations, many analysis techniques were developed to probabilistically verify Boolean as well as other discrete functions with a negligible probability of error. For example, by decomposing a circuit into regions which have mutually disjoint variable support set, and using such arithmetic transforms it was shown that an n-bit ALU requiring $\Theta(n^2)$ time using ROBDDs requires only linear resources with the probabilistic method [29].

In [29] some other methods for exploiting Boolean function properties for efficient hashing were also discussed. For example, it was shown that if the space of each function is partitioned into mutually disjoint subspaces then the hash code of the function corresponding to each partition can be calculated independently; the hash code of the function is the sum of the hash codes of individual partitions. This implies that to check if $H_\rho(f) = H_\rho(g)$, we can partition and hash both f and g independently. We do not need to keep both $\tilde{f}_i$ and $\tilde{g}_i$ in the memory at the same time. Further, it is not necessary that both f and g have the same window functions. The effectiveness of this technique was shown on special classes of functions like HWB in [29]. The techniques presented in [44] are directly applicable to probabilistic verification as well and provide automatic ways to generate such partitions.

Another technique called *collapse-with-compose* [29] allows efficient hashing of functions when orthogonal partitions cannot be easily found. This algorithm generates the hash code of the function directly from a decomposed representation without having to build the monolithic ROBDD of the output. For many difficult circuits in ISCAS-85 benchmark circuits, it was shown in [29] that this method can significantly outperform the monolithic ROBDD methods.

2.8 Verification of Arithmetic Circuits

So far we have discussed methods to compare two logic circuits at the bit-level. For many arithmetic circuits this may not be a desirable thing. First, the BDD data structure that is used for bit level verification may grow exponentially with the size of the circuit (for example, integer multiplication). Secondly, even if we can guarantee that the two netlists are equivalent that doesn't necessarily imply that the circuit is implementing the correct specification. To overcome these limitations of bit-level verification, a different approach for verifying arithmetic circuits was proposed in [38, 16]. Here a logic circuit represented as a vector of Boolean functions $\vec{f}$ is compared against the specification which is represented as a word level function F. The basic idea of this methodology is illustrated in Figure 3 [16]. Here the primary inputs are grouped into two sets, $\vec{x^1}$ and $\vec{x^2}$. For each set $\vec{x^i}$, we are given an encoding function ENC_i describing a word-level interpretation of the signals, e.g., unsigned binary, two's complement, BCD etc. The logic network is described as a set of Boolean functions given by $\vec{f}(\vec{x^1}, \vec{x^2})$ and an encoding, ENC_o for the outputs. The specification is given as a word-level function $F(X_1, X_2)$. The problem consists of verifying the following equation:

$$ENC_o(\vec{f}(\vec{x^1}, \vec{x^2})) = F(ENC_1(\vec{x^1}), ENC_2(\vec{x^2})) \quad (3)$$

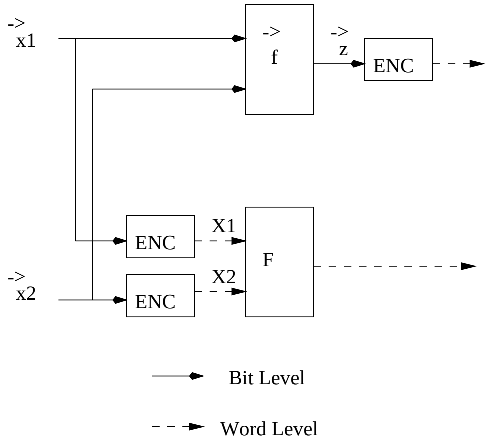

Figure 3: Word-Level Verification

For the word-level verification problem we need a data structure which can efficiently represent both bit-level and word-level functions. Many data structures have been proposed for this purpose, e.g., MTBDDs [19], ADDs [4], EVBDD [38], *BMD [16], Hybrid Decision Diagrams (HDDs) [18]. Out of these, the *BMD and the HDD data structures are of particular interest as they can represent integer multiplication efficiently. However, the verification strategy presented in [16] can not take advantage of this fact. This is because in the verification methodology, a bit level representation of the multiplier has to be created first which is then translated into the word-level representation. To circumvent this problem, [16] proposes a hierarchical verification strategy. This strategy requires a well-defined structure for the multiplier which has to be known *a priory*. This manual intervention somewhat reduces the appeal of both *BMDs and HDDs. In [26] a heuristic to efficiently construct *BMDs is presented which works well for integer multipliers but unfortunately not for other circuits like dividers and exponentiation.

[24] proposes a verification method which uses the recurrence equations of various arithmetic circuits such as multipliers, square functions, cube functions etc. to verify them. For example, a multiplier satisfies the recurrence equation, $f(x+1, y) = f(x, y) + y$ where $f(x, y) = xy$. Thus, to prove $f(x, y)$ represents a multiplier, we need to prove $f(x+1, y) = f(x, y) + y$, where x, y are input vectors for given circuit. Each side of the equation is represented as a separate circuit, and both circuits are efficiently compared by techniques such as such as [40, 30] which exploit the fact that the given circuits have very similar internal structures. As only a multiplier obeys the above recurrence relation, we can verify that the given circuit is indeed a multiplier without needing to represent the specification.

3 Combinational Verification Using Internal Correspondences

Typically during synthesis at the gate level, the designer makes local modifications to the logic network for adjusting timing, area, delay and other characteristics. Since the changes that are made to the networks are localized in nature, most of the original network remains structurally unchanged. Various techniques have been developed which exploit the internal correspondences in the two circuits to speed up the process of verification. We will first explain the terminology used in the rest of this section and then describe some of the methods that make use of the internal correspondences in combinational verification.

3.1 Terminology

We assume that the reader is familiar with the basic definitions of Boolean networks, fanins, fanouts, transitive fanins (TFIs) and transitive fanouts (TFOs) [11].

Definition 1 *Given a Boolean network $\eta = (V, E)$, a* **cutset** *in the TFI of a node $v \in V$ is defined as a set $S = \{v_1, v_2, \ldots, v_k\}$ of nodes in the TFI of v such that any path from the primary inputs to v must pass through a node $v_i \in S$. The size of the cutset is k.*

Definition 2 *A gate G is said to be* **unjustified** *under the present set of value assignments (preassigned values) to some of its fanin and fanout signals (preassigned signals) if among the assignments possible on the remaining signals there are (a) multiple assignments which are consistent with the preassigned signals, and (b) at least one assignment which can produce value at some signal that is inconsistent with its preassigned values.*

Examples of unjustified gates are shown in Figure 4. For instance, in the AND gate, suppose the output has already been assigned a Boolean value of 0. Now, if an assignment $\{a = 1, b = 1, c = 1\}$ is made, there is a conflict at this gate. Similarly, there exist multiple assignments that can be made at its inputs such as $\{a = 0, b = X, c = X\}$ for which the output value 0 can be produced. Thus, this is an example of an unjustified gate. Similarly, the reader can observe that for the output $c = 0$, the XOR gate in Figure 4 represents an unjustified gate.

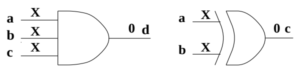

Figure 4: *Example of Unjustified Gates*

Definition 3 *Given an initial assignment of Boolean values to a subset of nodes in a Boolean network η, the process of determining the Boolean values at other nodes in η using the connectivity information of the nodes and the truth tables of the Boolean functions at the nodes is called* **direct implication**. *Boolean values at nodes that cannot be derived by direct implication but can be derived using the law of contrapositum [52] involve* **indirect implication**. *The process of deriving indirect implications is called* **learning**.

Figure 5 shows examples of a direct implication and an indirect implication. In the circuit on the left, if we set signal $a = 1$ then by a simple simulation (looking only at circuit connections, and the functionality of each circuit node) we can conclude that the signal f is forced to be a 0. Hence, $a = 1 \rightarrow f = 0$ is called a *direct* implication. Similarly for the circuit on the right, by law of contrapositum we can deduce that $f = 1 \rightarrow a = 0$. However, the reader can check that such a deduction could not have been arrived at by a naive simulation process such as the one used to derive the direct implication; thus, $f = 1 \rightarrow a = 0$ is an *indirect* implication.

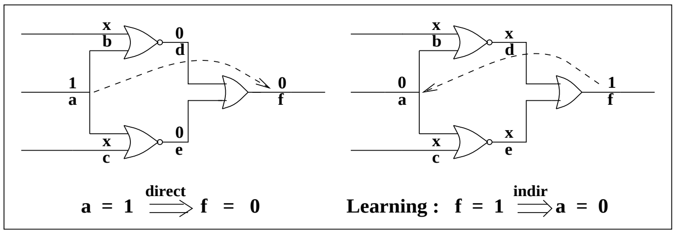

Figure 5: *Indirect Implication*

3.2 Learning Techniques: Techniques for Detecting Indirect Implications

There are several verification methods that extract and use internal correspondences between two given networks using learning based methods. Learning involves the extraction of indirect implications between nodes in a circuit. Recursive Learning (RL) [37], and Functional Learning (FL) [42, 30] are two of the more popular learning techniques. The concepts of FL and RL are illustrated below by means of an example.

Consider the circuit shown in Figure 6. Suppose that a Boolean 1 is injected at gate G. We wish to learn what $G = 1$ implies at gate H. Let us choose a cutset $\lambda = \{a, b, c\}$ of internal gates in the circuit. In FL, we build the ROBDDs for gates G and H based on λ. The ROBDDs for G and H are shown in the Figure 6. From these two BDDs we can infer that $G = 1 \Rightarrow H = 0$. Also, note that this relation between G and H cannot be derived by using direct implication. This is because after a Boolean 1 is injected at G, it becomes *unjustified*. Therefore, direct implication based simulation due to $G = 1$ cannot result in Boolean values being implied at any other gate in the circuit. However, $G = 1 \Rightarrow H = 0$ can be derived by carrying out the Boolean operation: $G \wedge \overline{H}$ and examining the resulting ROBDD for equivalence with G. The time complexity of functional learning is controlled by the size of BDDs and is impractical when the BDDs grow quite large.

In RL we note that $G = 1$ can be satisfied by either $H = 0, I = X$ or $H = X, I = 0$. However, $I = 0$ implies that $b = 1$ which in turn implies that $H = 0$. Therefore, $G = 1$ implies $H = 0$. Here, given a value assignment in the circuit, the deduction process recursively analyzes the effect of each justification vector, and intersects the common "effect" of every justification vector that can satisfy the given circuit condition. The result of this intersection process is the implication of the original value assignment in the circuit. The time complexity of recursive learning is exponential in the number of recursion levels, and in practice is limited to two or three levels of recursions.

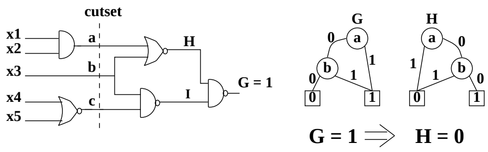

Figure 6: *Example of Functional Learning*

3.3 Techniques for Exploiting Internal Equivalences

There are several verification methods that exploit internal correspondences between two given networks. Berman *et al.* [5] proposed a technique to use internal equivalences in order to establish the functional equivalence of two networks. A min-cut based algorithm for decomposing networks was proposed to break down the entire verification problem into smaller subproblems. However, this method was plagued with the problem of *false-negatives*. A false negative refers to a situation where although the two functions which are being compared are equivalent, the verification algorithm incorrectly classifies them as different. Figure 7 shows an example of false-negative. F and G are equivalent outputs (both being equal to b) and node $d1$ is functionally equivalent to node $d2$. However, if a verification of F and G is attempted in terms of the cutsets shown by dotted lines in Figure 7, F will turn out to be inequivalent to G.

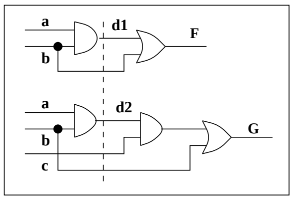

Figure 7: *False Negative*

Cerny and Mauras [17] introduced the notion of *cross-controllability* and *cross-observability* among the internal nodes on the appropriate cutsets in the two given networks to check for equivalence. In this, the functionality of the outputs was projected on to the inputs by successively projecting it to intermediate cutsets. However, no systematic algorithm for choosing an "appropriate" cutset was presented.

Brand [10] proposed an ATPG based technique to determine equivalences between the internal nodes in two given circuits. This method can find nodes which are equivalent under the observability don't care (ODC) set. Given two circuits C_1 and C_2, and two potentially equivalent nodes $n_1 \in C_1$ and $n_2 \in C_2$, a new XOR gate Y, is introduced in C_1 such that n_1 and n_2 are the two fanins of Y.[1] Fig. 8 shows an example of such a *miter* circuit. The dotted lines show the original circuit connections. A new XOR gate is introduced and the outputs of the XOR gate feeds the nodes that were originally fed by node n_1. Next, an ATPG tool is used to test the fault s-a-0 at Y as shown in the figure. If the fault is proved to be redundant then n_2 can replace n_1 in circuit C_1. However, this method of finding equivalent nodes can become inefficient if the faults that are

[1] In verification algorithms, potential equivalent nodes between two circuits are typically decided by simulating the given circuits on some k simulation vectors. Two nodes are now called potentially equivalent if they have an equivalent output on each of the k vectors. The value of k can be decided dynamically or a priori.

tested to determine internal equivalences are intractable or if not too many internal equivalences exist in the two designs.

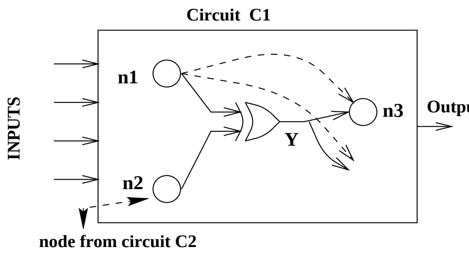

Figure 8: *Testing node equivalences by creating a miter*

Recently, several learning based techniques for combinational verification have been proposed. In [36], a combinational verification tool, HANNIBAL, based on recursive learning was presented. HANNIBAL operates in two distinct phases. In the first phase, learning is carried out at all the nodes in the two networks for a user specified number of learning levels; often, this phase itself can verify several primary outputs of the two networks. In the second phase, using the learning information derived in the first phase, an ATPG tool is invoked for verifying the remaining primary outputs. In [41] another verification tool, VERIFUL, was presented which is based on functional learning. This tool also has two phases like HANNIBAL. Here learning is carried out at each gate g using ROBDDs. These ROBDDs are built using a cutset that is at a structural distance[2] d away from g. Here d can vary from 1 to a predetermined maximum distance d_{max}. The amount of learning obtained in a network can be increased by increasing d_{max}. However, the sizes of the ROBDDs that are built usually increase with the increase in d. This results in an increase in the time and space resources required. Two other learning based verification algorithms were presented in [30, 48]. Both of these methods consist of an initial learning phase followed by an ROBDD based equivalence-checking phase. Methods to reduce the ROBDD sizes using the learning information were presented. In [48] the ROBDDs were pruned with an ATPG tool that uses the learning information derived in the first phase. In [30] *invariants*[3] based on learnings are used to simplify the ROBDDs. This technique also successively composes BDDs in terms of cutsets of internal equivalent gates till the functional equivalence is resolved.

The learning based techniques have several limitations. First, they are unable to derive *all* internal equivalences in limited computational resources. All the known learning techniques discover equivalences between internal gates in circuits using two indirect implications: to find if $f \equiv g$ they individually determine if $f \Rightarrow g$, and then if $g \Rightarrow f$. However, finding indirect implications, whether through ROBDD operations [41] or techniques such as recursive learning [37], can be relatively expensive. Another problem is that there is no simple method to determine, a priori, the number of levels of learning that will be required on a given pair of circuits. Hence, a complete automation of learning based verification tools may be difficult.

Recently another efficient technique that analyzes internal similarities between circuits using ROBDDs was proposed in [40]. Beginning from gates closest to primary inputs, function $(g \oplus h)$ is calculated for all potentially equivalent gate pairs (g, h) using ROBDDs. The equivalent gates between two circuits are merged as shown in Figure 8. The ROBDDs are built using internal variables which are usually introduced at gates that have already been shown to be equivalent. The set of internal variables (gates) is chosen such that we minimize gates that have a path to another gate in the same set. This technique gives up to an order of magnitude speed-up over [48] on many benchmark circuits.

[2] The structural level in functional learning of [41] is a close analogue of the level of learning in recursive learning.

[3] If $a \Rightarrow b$, where a and b are two nodes, then $\overline{a} + b$ is an invariant.

4 Conclusion

Due to the memory explosion problem, BDDs alone appear unsuitable for verifying large designs. However, they form a crucial representation vehicle for the internal correspondence based verification techniques. A practical combinational verification tool must consolidate diverse techniques for extracting internal correspondences. Such a technique must use the state of the art BDDs, ATPG, as well as implication based techniques. For example, it has been observed that a verification technique based on exploiting internal equivalences can fail on circuits that have relatively few equivalent nodes. Therefore, such a technique needs to be combined with a learning algorithm to make use of the indirect implication relations that exist between the nodes of the two circuits. To verify inequivalent circuits or internal nodes, use of ATPG techniques appears essential. Finally, in cases where both internal equivalence and learning techniques prove inadequate, verification techniques should be augmented by functional partitioning, possibly using representations such as partitioned-ROBDDs.

5 Acknowledgement

We would like to thank Rajarshi Mukherjee for his assistance with this paper. The second author was supported by CA State MICRO program grant #94-110 and SRC 95-DC-324.

References

[1] Sheldon B. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, C-27:509–516, June 1978.

[2] P. Ashar and M. Cheon. Efficient breadth-first manipulation of binary-decision diagrams. *ICCAD*, 1994.

[3] P. Ashar, A. Ghosh, and S. Devadas. Boolean satisfiability and equivalence checking using general binary decision diagrams. *ICCD*, 1991.

[4] R. Bahar *et. al.* Algebraic decision diagrams and their applications. *ICCAD*, 1993.

[5] C. L. Berman and L. H Trevyllian. Functional comparison of logic designs for vlsi circuits. *ICCAD*, 1989.

[6] J. Bern, C. Meinel, and A. Slobodova. Efficient OBDD-Based Boolean Manipulation in CAD Beyond Current Limits. *DAC*, 1995.

[7] J. Bern, C. Meinel, and A. Slobodova. Some Heuristics for Generating Tree-like FBDD Types. *IEEE Transactions on Computer-Aided Design*, pages 127–134, January 1996.

[8] M. Blum *et. al.* Equivalence of free Boolean graphs can be decided probabilistically in polynomial time. *Information Processing Letters*, 10:80–82, March 1980.

[9] K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient Implementation of a BDD Package. *DAC*, 1990.

[10] D. Brand. Verification of large synthesized designs. *ICCAD*, 1993.

[11] R. K. Brayton, R. Rudell, A. L. Sangiovanni-Vincentelli, and A. R. Wang. MIS: A Multiple-Level Logic Optimization System. *IEEE Transactions on Computer-Aided Design*, CAD-6(6):1062–1081, November 1987.

[12] R. E. Bryant. Graph-based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35:677–691, August 1986.

[13] R. E. Bryant. Boolean Analysis of MOS Circuits. *IEEE Transactions on Computer-Aided Design*, pages 634–649, July 1987.

[14] R. E. Bryant. Extraction of gate level models from transistor circuits by four-valued symbolic analysis. *ICCAD*, 1991.

[15] R. E. Bryant. Symbolic boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys*, 24:293–318, September 1992.

[16] R. E. Bryant and Y. Chen. Verification of arithmetic circuits with binary moment diagrams. *DAC*, 1995.

[17] E. Cerny and C. Mauras. Tautology checking using cross-controllability and cross-observability relations. *ICCAD*, 1990.

[18] E. M. Clarke, M. Fujita, and X. Zhao. Hybrid decision diagrams. *ICCAD*, 1995.

[19] E. M. Clarke *et. al.* Spectral transforms for large boolean functions with applications to technology mapping. *DAC*, 1993.

[20] R. Drechsler *et. al.* Efficient representation and manipulation of switching functions based on Ordered Kronecker Functional Decision Diagrams. *DAC*, 1994.

[21] S. Malik *et. al.* Logic Verification using Binary Decision Diagrams in a Logic Synthesis Environment. *ICCAD*, 1988.

[22] L. Fortune *et. al.* The complexity of equivalence and containment for free single variable program schemes. *Lecture Notes in Computer Science 62, Springer-Verlag*, pages 227–240, 1978.

[23] M. Fujita, H. Fujisawa, and N. Kawato. Evaluation and Improvements of Boolean Comparison Method Based on Binary Decision Diagrams. *ICCAD*, 1988.

[24] M. Fujita. Verification of Arithmetic Circuits by Comparing Two Similar Circuits. *CAV*, 1996.

[25] J. Gergov and C. Meinel. Efficient Boolean Manipulation With OBDD's can be Extended to FBDD's. *IEEE Transaction on Computers*, 43(10):1197–1209, 1994.

[26] K. Hamaguchi, A. Morita, and S. Yajima. Efficient construction of binary moment diagrams for verifying arithmetic circuits. *ICCAD*, 1995.

[27] J. Jain, J. Bitner, M. Abadir, D. S. Fussell, and J. A. Abraham. Indexed BDDs: Algorithmic advances in techniques to represent and verify Boolean functions. *To be published in IEEE Transactions on Computers.*

[28] J. Jain, J. Bitner, D. S. Fussell, and J. A. Abraham. Functional partitioning for verification and related problems. *Brown/MIT VLSI Conference*, 1992.

[29] J. Jain, J. Bitner, D. S. Fussell, and J. A. Abraham. Probabilistic verification of Boolean functions. *Formal Methods in System Design*, 1: 61 – 115, 1992.

[30] J. Jain, R. Mukherjee, and M. Fujita. Advanced Verification Techniques Based on Learning. *DAC*, 1995.

[31] J. Jain, A. Narayan, C. Coelho, S. Khatri, A. Sangiovanni-Vincentelli, R. Brayton, and M. Fujita. Decomposition Techniques for Efficient ROBDD Construction. *Formal Methods in CAD 96*, LNCS. Springer-Verlag, 1996.

[32] S.-W. Jeong, B. Plessier, G. Hachtel, and F. Somenzi. Structural BDDs: Trading canonicity for structure in verification algorithms. *ICCAD*, 1991.

[33] T. Kam and P. A. Subrahmanyam. Comparing Layouts with HDL Models: A Formal Verification Technique. *IEEE Transactions on Computer-Aided Design*, pages 503–509, April 1995.

[34] U. Kebschull *et. al.* Multilevel logic synthesis based on Functional Decision Diagrams. *European DAC*, 1992.

[35] A. Kuehlmann, A. Srinivasan, and D. P. LaPotin. A Formal Verification Program for Custom CMOS Circuits. *IBM Journal of Research and Development*, January 1995.

[36] W. Kunz. HANNIBAL: An Efficient Tool for Logic Verification Based on Recursive Learning. *ICCAD*, 1993.

[37] W. Kunz and D. K. Pradhan. Recursive learning: An attractive alternative to the decision tree for test generation in digital circuits. *ITC*, 1992.

[38] Y-T Lai and S. Sastry. Edge-valued binary decision diagrams for multi-level hierarchical verification. *DAC*, 1992.

[39] C. Y. Lee. Representation of switching circuits by binary-decision programs. *Bell Syst. Tech. J.*, 38:985–999, 1959.

[40] Y. Matsunaga. An Efficient Equivalence Checker for Combinational Circuits. *DAC*, 1996.

[41] R. Mukherjee, J. Jain, and M. Fujita. VERIFUL: VERIfication using FUnctional Learning. *EDAC*, 1995.

[42] R. Mukherjee, J. Jain, and D. K. Pradhan. Functional Learning: A new approach to learning in digital circuits. *IEEE VLSI Test Symp.*, 1994.

[43] A. Narayan, S. P. Khatri, J. Jain, M. Fujita, R. K. Brayton, and A. Sangiovanni-Vincentelli. A Study of Composition Schemes for Mixed Apply/Compose Based Construction of ROBDDs. *Intl. Conf. on VLSI Design*, 1996.

[44] A. Narayan, J. Jain, M. Fujita, and A. L. Sangiovanni-Vincentelli. Partitioned-ROBDDs - A Compact, Canonical and Efficiently Manipulable Representation for Boolean Functions. *ICCAD*, 1996.

[45] H. Ochi, K. Yasouka, and S. Yajima. Breadth-first manipulation of very large binary-decision diagrams. *ICCAD*, 1993.

[46] S. Panda and F. Somenzi. Who Are the Variables in Your Neighborhood. *ICCAD*, 1995.

[47] S. Panda, F. Somenzi, and B. Plessier. Symmetry Detection and Dynamic Variable Ordering of Decision Diagrams. *ICCAD*, 1994.

[48] S. M. Reddy, W. Kunz, and D. K. Pradhan. Novel Verification Framework Combining Structural and OBDD Methods in a Synthesis Environment. *DAC*, 1995.

[49] R. L. Rudell. Dynamic Variable Ordering for Ordered Binary Decision Diagrams . *ICCAD*, 1993.

[50] J. V. Sanghavi, R. K. Ranjan, and A. Sangiovanni-Vincentelli, and R. K. Brayton. High Performance BDD Package by Exploiting Memory Hierarchy. *DAC*, 1996.

[51] K. J. Singh and P. A. Subrahmanyam. Extracting RTL models from transistor netlists. *ICCAD*, 1995.

[52] D. F. Stantat and D. A. McAllister. *Discrete Mathematics in Computer Science*. Intl. Series in Applied Mathematics. Prentice-Hall, Englewood Cliffs, N.J., 1977.