Novel Verification Framework Combining Structural and OBDD Methods in a Synthesis Environment*

Subodh M. Reddy†
Laboratory for Digital and
Computer Systems Research
Department of Computer Science
Texas A&M University
College Station, Texas 77840

Wolfgang Kunz
Fault Tolerant
Computing Laboratory
Max-Planck-Society
University of Potsdam
14415 Potsdam, Germany

Dhiraj K. Pradhan
Laboratory for Digital and
Computer Systems Research
Department of Computer Science
Texas A&M University
College Station, Texas 77840

Abstract—This paper presents a new methodology for formal logic verification for combinational circuits. Specifically, a structural approach is used, based on indirect implications derived by using Recursive Learning. This is extended to formulate a hybrid approach where this structural method is used to reduce the complexity of a subsequent functional method based on OBDDs. It is demonstrated how OBDD-based verification can take great advantage of structural preprocessing in a synthesis environment. The experimental results show the effective compromise achieved between memory-efficient structural methods and functional methods. One more advantage of these methods lies in the fact that resources that go into logic synthesis can effectively be reused for verification purposes.

I. INTRODUCTION

Traditionally formal logic verification for combinational circuits has been attempted by either a purely functional, e.g., [7, 6] or a purely structural approach [20, 11, 4], though the former is more common. As shown in [11, 4], structural approaches to logic verification can perform extremely well, if the circuits have some structural "similarity"; however, they can fail otherwise. OBDD-based verification, on the other hand, is independent of the structural representation of the individual circuits, but the construction of the OBDDs can be highly memory-expensive. The goal of this paper is to propose a general scheme to combine such structural and functional approaches for logic verification, to mutually exploit the advantages of both the paradigms.

Current synthesis tools are comprised of numerous different steps involved in circuit transformations. As pointed out in [11, 1], synthesis is an incremental process consisting of many small operations. Therefore, along the synthesis process, subsequent circuit designs can be expected to have some "similarity", and any tool able to efficiently exploit such similarity can perform well. Furthermore, design errors are often introduced by interference of the human designer. Therefore, it is very important for the designer to have efficient tools for checking the functional correctness of the design. This is the motivation for our research and we propose an efficient method to reduce the complexity of BDD-based verification, by making use of the similarity between designs as it can be expected in many practical verification problems.

The underlying philosophy of this paper is as follows: using structural methods to capture the similarity between circuits and to identify sub-circuits, by partitioning the original circuits based on the partitioning criterion as stated below; and to use a functional approach to prove the equivalence of these subcircuits. This is outlined in Figure 1. Also it is desirable to keep alternating between these methods incrementally until a solution is obtained.

Let C1 and C2 be the circuits to be verified. Assume that both circuits are cut vertically, as shown in Figure 2, so that both circuits are split into two parts. Let C1' and C2' be the circuit partitions at the primary outputs of the original circuits. C1' and C2' are shown in Figure 3.

Criterion for circuit partitioning: A cut through circuit C1 and C2 is *permissible* if the functional equivalence of the partitioned circuits C1' and C2', implies the equiva-

*Research reported supported in part by NSF grant MIP–9406946 and ONR grant #N00014–92–J–1366.

†The author is currently at the Unisys Corporation; this research was conducted while visiting Max-Planck Society, Potsdam and is included in his M.S. thesis at Texas A&M University, May 1994.

32nd ACM/IEEE Design Automation Conference ®

lence of circuits C1 and C2.
The main problem of this approach is *false negatives* [2], explained in the next Section.

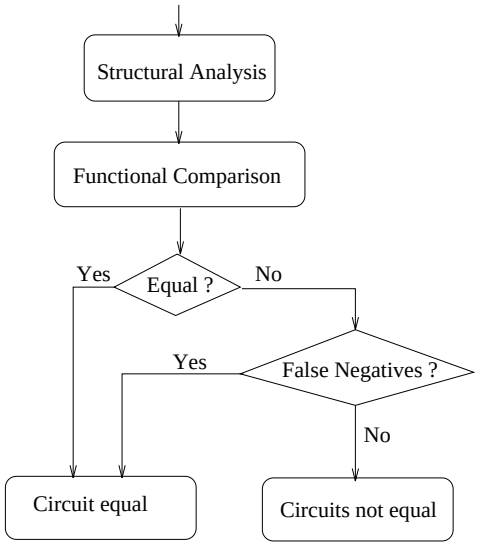

Figure 1: A hybrid verification tool

A. False Negatives

The criterion for circuit partitioning, that the original circuits are equivalent if the partitioned circuits are equivalent, does not guarantee that the opposite is always true. It is possible that the original circuits are equivalent but the partitioned sub-circuits are not. This problem is generally referred to as *false negatives* [2].

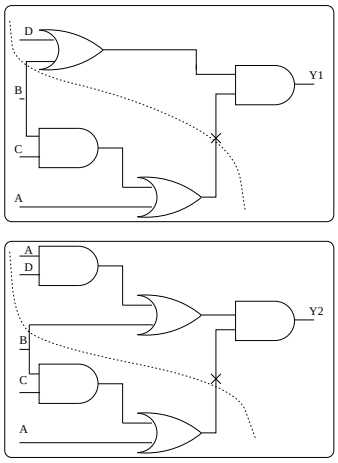

Figure 2: The *False Negative Problem:* Original Circuits C1 and C2

Consider the circuits in Figure 2. It is obvious that the two circuits are equivalent. Now, consider the partition, shown by a dashed line. This satisfies our criterion for partitioning. The cut circuits are shown in Figure 3. These circuits are not equivalent even though the original circuits

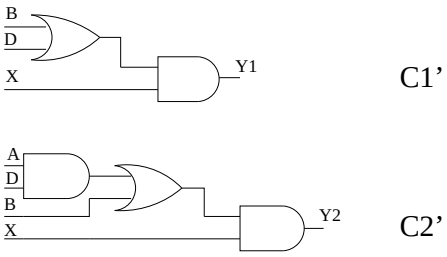

Figure 3: The *False Negative Problem:* Partitioned Circuits C1' and C2'

are equivalent. Therefore, efficient techniques have to be incorporated to deal with this problem.

B. Partitioning the Circuit

For partitioning the circuits, we make use of the internal equivalencies derived from the method in [11]. These internal equivalencies form an excellent basis for circuit partitioning and they satisfy our criterion. However, it must be noted that any other method which satisfies our criterion for circuit partitioning can be used, instead. Similarly, as in [4], it is promising to use the concept of permissible functions to identify larger sets of possible cuts. This can be accomplished by using the notion of D-implications as in [12].

Our method can be outlined as follows: Recursive Learning [13] is used to find implications between signal values from which internal equivalent points are extracted. The circuit is partitioned by cutting the circuit through the internal equivalencies, as shown by the dashed line in Figure 2. To obtain optimum benefit, we always attempt to cut the circuit as close to the primary outputs as possible. The internal equivalencies are treated as new, independent "pseudo-inputs" of the reduced circuit. Once the circuits are cut, the primary outputs of the circuits are compared for equivalence. Karl Brace's OBDD package[3] was used for building OBDDs. Our experiments suggest that *false negatives* are not uncommon, the principle reason being the inter-dependencies of these pseudo-inputs. The BDDs thus formed may contain some combinations of pseudo-inputs which are inconsistent in the original circuit and hence, represent a don't care set for the partitioned circuit. Therefore, the thrust of this paper is dedicated to addressing this.

II. DESCRIPTION OF THE ALGORITHM

The block diagram of the program flow is shown in Figure 4. The algorithm consists of two stages:

1. Structural analysis (Identification of internal equivalent signals)

2. Functional Comparison (Building OBDDs for the outputs and checking for their equivalence)

A. Structural analysis

The indirect implications which lead to the identifica-

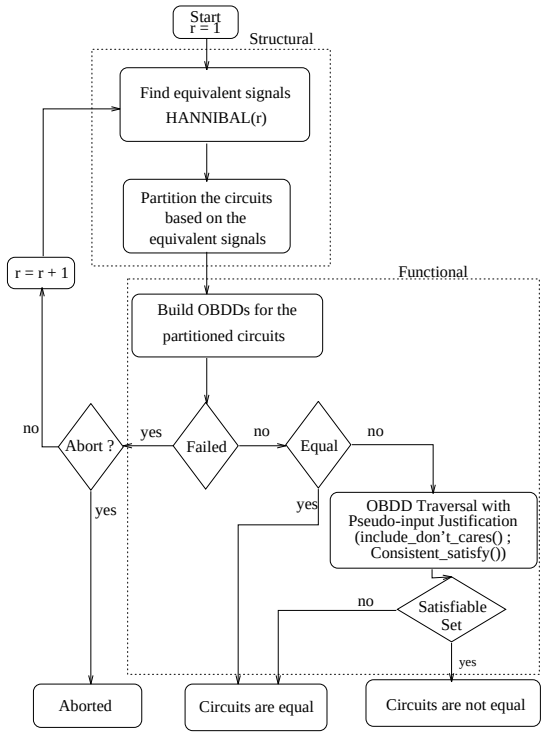

Figure 4: Block diagram of the program flow

tion of equivalent points are found using Recursive Learning [13]. Recursive learning is a complete algorithm to perform implications [13] in a combinational circuit. (refer [13] for a complete description). We use the verification tool HANNIBAL [11] for this purpose, which is based on implications and learning. The two circuits to be verified are first joined at the inputs, their respective outputs then tied together by an exclusive-or, the output of which feeds an or-gate.

At every signal of this combined circuit, called *miter* in [4], a logical one and a logical zero are assigned and their logic consequences noted with the help of Recursive Learning. Let A and B be two signals in a circuit. If an assignment $A = 0$ and $A = 1$ implies that $B = 0$ and $B = 1$, respectively, then these two signals are functionally equivalent. The number of internal equivalent points found this way depends on the *depth of recursion* [11] used. We use this as a preprocessing step. First, a recursion depth of one is used and if this proves to be insufficient, then it is incremented and the process is repeated.

B. Functional Comparison

First, we need to identify the sub-circuits by partitioning the original circuits into smaller sub-circuits. For this, a simple procedure is used, tracing from the outputs towards the inputs. Using the classical depth first search from the outputs, we trace until an equivalent point or a primary input is encountered and marked. All these marked signals are treated as independent pseudo-inputs to the traced part of the circuit containing the outputs; i.e., these marked signals are the desired partitions. This is performed on one circuit and the partition is mapped into the other circuit to the corresponding equivalent points. In some cases, this mapping may not result in a complete cut in the other circuit; in these cases, the outputs whose cone does not contain a complete cut is traced in a similar fashion, and the cut is made complete.

The OBDDs for the outputs are calculated for each circuit in terms of these pseudo-inputs that identify the respective circuit's partitions. The OBDDs are built, using *Apply* operation [6] by traversing the circuit from the pseudo-inputs towards primary outputs and building intermediate, temporary OBDDs at each node's output. The respective OBDDs of the output are compared for equivalence. If they are equivalent, the circuits are proven to be equivalent and if they are not equivalent, then it has to be examined whether this is a false negative.

Pseudo-input justification

To check for false negatives, we propose the following approach. First, a new OBDD is formed by first Exor-ing the respective output OBDDs, and then forming an OR of all these Exor-ed OBDDs. Then, the following procedure is applied:

```
Consistent_satisfy(bdd_node var)
{
    if( var == constant ONE )
    {                   /*Found consistent satisfiable set*/
        if(Justifiable)
    /* Found a distinguish vector */
                return ONE
        else
                /* Continue with the traversal */
                return ZERO
    }
    else if ( var == constant ZERO )
        /* Continue with the traversal */
        return ZERO
    Assign: '1' to the node which represents
    the variable "var" represents
    if(Imply() == consistent)
        if(Consistent_satisfy(var.high))
                return ONE
    Erase this Assignment and its implications
    Assign: '0' to the node representing the variable "var"
    if(Imply() == consistent)
        if(Consistent_satisfy(var.low))
                return ONE
    Erase this Assignment and its Implications
    return ZERO
}
```

This is a process where, first, a consistent, satisfiable set is found by traversing the exor-ed OBDD and next,

an attempt is made to find a justification sequence at the primary inputs of the original circuit for this satisfiable set. If it cannot be justified, OBDD traversal is continued and a new consistent satisfiable set is found, the process is repeated until, either it is found that there is no consistent satisfiable set which can be justified, which means the circuits are equivalent, or a distinguishing vector is generated.

The recursive function *consistent_satisfy()* takes an OBDD node as an argument and finds a consistent satisfiable set. Function *justify()* is used to check whether the satisfiable sequence found in *consistent_satisfy()* is justifiable.

For the justification process (*justify()*), we use test generation techniques based on FAN's [10] multiple backtrace procedure and implicit enumeration. The prestored indirect implications are used to speed up the process. In our experiments *Consistent_satisfy*, generally proved efficient to solve the false negative problem. However, in many cases, the process can be speeded up considerably by the following technique which allows decrease of the size of the OBDD that has to be traversed by *Consistent_satisfy* .

Incorporating the don't cares

```
include_don't_cares(f(x1, x2, ..., xn))
{
    for(i = x1 to xn)
    {
        assign : i = v
        fi =| f|i=v          /*Divide f into co-factors*/
        fī =| f|i=v̄
        Imply() in the original circuit
        for(j = x1 to xn and j ≠ i)
        {
            if(circuit(j).output ≠ don't care)
                fi =| fi|j's value
        }
        assign : i = v̄
        Imply()                    /*Make implications*/
        for(j = x1 to xn and j ≠ i)
        {
            if(circuit(j).output ≠ don't care)
                fī =| fv̄|j's value
        }
        fnew = ITE(i, fi, fī)
        if(size(fnew < f))
            return fnew
        else
            return f
    }
}
```

We do an implication analysis to incorporate partial information of the don't care information into the OBDDs.

Circuit	Functional	Structural + Functional	
	OBDD size	Rec. Depth	OBDD Size
C432	55,023	1	7 *
C499	136,255	1	32*
C1355	136,255	1	32*
C1908	31,978	1	25*
C2670	Unable	1	166,665
C3540	Unable	2	3516
C5315	11659	1	1677
C6288	Unable	1	33*
C7552	Unable	1	160,510

Table 1: Comparison of final OBDD Sizes.

Circuit	Functional	Structural + Functional		
	OBDD [s]	Struct.[s]	Funct.[s]	Total [s]
C432	60.9	1.0	1.2	2.2
C499	89.32	1.9	0.27	2.17
C1355	143.6	6.6	0.73	6.73
C1908	30.44	11.2	3.34	14.54
C2670	Unable	8.7	150.6	159.3
C3540	Unable	52.8	14.84	67.64
C5315	20.52	32.4	340.4	372.8
C6288	Unable	21.5	11.24	32.74
C7552	Unable	97.2	5486.1	5583.3

Table 2: CPU times [s].

This procedure is listed above and is explained below. As pointed out, the cause of a false negative is the interdependency of the pseudo-inputs which means that if the equivalent points are independent, then no false negatives can occur. f is a function of n variables $x_1, x_2, ..., x_n$. First, f is divided into two cofactors f_i and $f_{\bar{i}}$, based on a variable, $i \in (x_1, x_2, ..., x_n)$. A signal value is assigned to the signal in the original circuit representing the variable i, and implication performed. If this implication results in the signals which represent other variables being specified, then "restrictions" are made on the respective cofactors, as shown in the listing. This process is continued for all the signals. Finally the "restricted" cofactors are combined by an ITE operator, as defined in [3].

This procedure, however, is not complete i.e., it does not find all possible don't care sets, but experimental results show that this considerably reduces the OBDD sizes. It is expected that our results can be further improved significantly using well-known methods for don't care extraction and OBDD minimization with respect to given don't care sets, e.g. [9].

III. RESULTS

To examine the benefit for OBDD-based verification from a structural preprocessing phase, we conducted a series of verification experiments on the ISCAS-85 bench-

Circuit	Total number of Outputs	# Outputs found equal by Structural Analysis	# Outputs with Isomorphic OBDDs	# Outputs with Different OBDDs (False Negatives)
C432	7	7	—	—
C499	32	32	—	—
C1355	32	32	—	—
C1908	25	25	—	—
C2670	140	126	6	8
C3540	22	6	6	10
C5315	123	58	49	16
C6288	33	33	—	—
C7552	108	56	42	10

Table 3: False Negatives.

marks. The ISCAS-85 benchmarks were verified against their non-redundant versions that are also available from MCNC. This verification experiment adequately reflects the range of applications we have in mind for our hybrid verification method. The circuits have been modified at several different locations, but there is still "similarity" between them, which can be expected to be the case for many practical verification problems, especially after engineering changes (ECs). As mentioned previously, we used Karl Brace's BDD package for our implementation [3]. Experiments were conducted on the MCNC ISCAS benchmark circuits to verify the equivalence of the redundant [5] and non-redundant [19] sets of these circuits. The prestored indirect implications (the internal equivalent points) are read from a file generated by HANNIBAL [11] as a preprocessing step. No special variable-ordering techniques are used for our BDD formation. BDD variables are created for each equivalent point and ordered, based on their output distance. The ordering is fixed for all the outputs of the circuits. The results are presented in Tables 1 and 2. Table 1 compares the *final* OBDD sizes for the whole and the cut circuits, respectively. The variables used for creating OBDDs for the whole circuit were also ordered, based on their output distance. In Table 2, the CPU time in seconds is listed. The recursion depth [11] used for preprocessing is also listed for each circuit in Table 1.

The sizes are the aggregate sizes for all the outputs which take sharing into account [3]. Importantly, in all examined cases, the BDD sizes shrink drastically after the structural preprocessing phase. For some circuits marked by an $*$, structural analysis with recursion depth one could complete the job, alone [11]; in these cases, the number of BDD nodes shown is just the number of variables that was created for outputs. For circuit c3540, we could not build a BDD for a preprocessing recursive depth of one, so the preprocessing is done with a recursive depth of two. In this way, more internal equivalencies are generated which, in turn, make the partitioned circuit tinier, causing the BDD sizes to shrink. This aptly demonstrates how structural and functional techniques can complement each other to provide more efficient means to solve the verification problem. Note that our results can further be improved drastically by applying more sophisticated ordering techniques, as have been reported in literature [16, 14, 15]. The BDD sizes for the examined circuits were extremely low, compared to all the conventional functional techniques. Consider the circuit c6288. As is well known, no optimal variable order exists and any OBDD based verification will fail. In this case, as demonstrated in [11], the preprocessing, itself, has proven that the circuits are equivalent, without a need for building an OBDD.

Table 3 lists the number of false negatives encountered for the benchmark circuits. The second column gives the number of outputs of each circuit. The number of outputs proven to be equivalent in the structural analysis alone are shown in the third column of this table. The fourth and fifth columns represent the number of outputs with isomorphic OBDDs and outputs with different OBDDs (i.e., false negatives), respectively.

The results so far presented were only for the circuits which were equivalent. It is interesting to see how our methods fare when the circuits are not equivalent. For this reason, we changed a gate in the benchmark circuits, which was picked randomly, to a different type, so that the functionality of the original circuit was changed. These modified circuits were compared with their original counterparts. Table 4 presents the results for these true negatives, which compares OBDD sizes and CPU time between the OBDD-based pure functional method and our hybrid approach. For all the cases, the depth of recursion used was one. In the majority of the cases, the inequivalence was proven in the structural stage, itself. This is because the structural techniques are particularly powerful in generating a distinguishing vector, without completely enu-

Circuit	Functional		Funct. + Struct.	
	OBDD Size	CPU Time [s]	OBDD Size	CPU Time [s]
c432	55047	54.01	0	1
c499	365153	193.9	512	59
c1355	177573	285.4	0	9
c1908	38538	23.27	0	10
c2670	Unable	—	0	11
c3540	Unable	—	10371	15
c5315	12889	25.29	0	26
c6288	Unable	—	0	23
c7552	Unable	—	0	97

Table 4: True negatives.

merating the search space. In the cases where OBDDs had to be created, the required sizes are very small and in all the cases, the CPU time is relatively low.

As mentioned before, further improvements can be expected by a more general structural phase using the concepts of [4, 12, 18], and/or by a more sophisticated functional phase using better variable orderings or other graph representations of Boolean functions [7, 17, 8]. Independent of such promising extensions, our research demonstrates how functional and structural methods for logic verification can be combined efficiently. Our experimental results confirm that a hybrid approach of structural and functional techniques provides a useful and flexible tool set to perform logic verification in a synthesis environment.

IV. CONCLUSION

In this paper, we presented the effectiveness of a hybrid logic verification tool based on both structural [11] and functional [6] techniques. This hybrid verification tool is based on OBDDs, the structural analysis based on recursive learning. This provides a means for an effective tradeoff between time and memory. An implication-based routine is developed for finding don't care information, which reduces the size of the OBDDs used.

References

[1] E. J. Aas, K. Klingsheim, and T. Steen. Quantifying design quality. In *Proceedings of EURO ASIC*, pages 172–177, 1992.

[2] C. L. Berman and L. H. Trevillyan. Functional comparison of logic designs for vlsi circuits. In *Intl. Test Conference*, pages 456–459, 1989.

[3] K. Brace. Efficient implementation of a bdd package. In *Design Automation Conference*, pages 40–45, 1990.

[4] D. Brand. Verification of large synthesized designs. In *Intl. Conference on Computer-aided Design*, pages 534–537, 1993.

[5] F. Brglez and H. Fujiwara. A neutral netlist of 10 combinational benchmark circuits and a target translator in fortran. In *Special Session on the 1985 IEEE Intl. Symposium on Circuits & Systems*, 1985.

[6] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, pages 677–691, Aug. 1986.

[7] J. Jain et al. Indexed bdds: Algorithmic advances in techniques to represent to represent and verify boolean functions. Technical Report UT-CERC-TR-JAA-93-02, Comp. Eng. Research Center, 1993.

[8] R. Drechsler et al. Efficient representation and manipulation of switching functions based on ordered kronecker functional decision diagrams. In *Design Automation Conference*, pages 415–419, 1994.

[9] T.R. Shiple et al. Heuristic minimization of bdds using don't cares. In *Design Automation Conference*, pages 225–231, 1994.

[10] H. Fujiwara and T. Shimono. On the acceleration of test generation algorithms. In *Intl. Symposium on Fault-tolerant Computing*, pages 98–105, 1983.

[11] W. Kunz. Hannibal: An efficient tool for logic verification, based on recursive learning. In *Intl. Conference on Computer-aided Design*, pages 538–543, 1993.

[12] W. Kunz and P. Menon. Multi-level logic optimization by implication analysis. In *Intl. Conference on Computer-aided Design*, pages 6–13, November, 1994.

[13] W. Kunz and D. K. Pradhan. Recursive learning: A new implication technique for efficient solutions to CAD problems – test, verification, and optimization. In *IEEE Trans. on Computer-aided Design of Integrated Circuits and Systems*, Vol. 13, No. 9, pages 1143–1157, September, 1994.

[14] S. Malik, A. Wang, R. Brayton, and A. Sangiovanni-Vincentelli. Logic verification using binary decision diagrams in a logic synthesis environment. In *Intl. Conference on Computer-aided Design*, pages 6–9, Nov. 1988.

[15] M. R. Mercer, R. Kapur, D. E. Ross. Functional approaches to generate orderings for efficient symbolic representations. In *Design Automation Conference*, 1992.

[16] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *Intl. Conference on Computer-aided Design*, pages 42–47, Nov. 1993.

[17] A. Shen, S. Devadas, and A. Ghosh. Probabilistic construction and manupulation of free boolean diagrams. In *Intl. Conference on Computer-aided Design*, 1993.

[18] D. Stoffel, W. Kunz, S. Gerber. Multi-level Logic Synthesis by And-Or-Graphs. Technical Report, MPI-I-95-602, Max-Planck Fault-Tolerant Computing Group, 1995.

[19] G. J. Tromp and A. J. van de Goor. Logic synthesis of 100-percent testable logic networks. In *Intl. Conference on Computer-aided Design*, 1991.

[20] R. Wei and A. L. Sangiovanni-Vincentelli. Proteus: A logic verification system for combinational circuits. In *Intl. Test Conference*, 1986.