An Efficient Equivalence Checker for Combinational Circuits

Yusuke Matsunaga
FUJITSU LABORATORIES LTD, Kawasaki 211-88, Japan

Abstract— This paper describes a novel equivalence checking method for combinational circuits, which utilizes relations among internal signals represented by binary decision diagrams. To verify circuits efficiently, A proper set of internal signals that are independent with each other should be chosen. A heuristic based on analysis of circuit structure is proposed to select such a set of internal signals. The proposed verifier requires only a minute for equivalence checking of all the ISCAS'85 benchmarks on SUN-4/10.

1 Introduction

Although equivalence checking of given two combinational circuits is one of the important problems in CAD for logic design, it is known to be a co-NP complete problem. This means there is no hope to develop a complete algorithm that can solve any equivalence checking problem efficiently. Therefore, developing practically good heuristics for equivalence checking is needed.

Recently, several BDD based methods and ATPG based methods for equivalence checking have been proposed [1, 2, 3, 4, 5, 6, 7, 8]. They perform very efficiently for some examples, however, for a couple of examples with thousands of gates, they fail or take hours to verify. In this paper, we propose a more robust and efficient equivalent checking method for combinational circuits, which is based on BDD manipulation. Experiments of equivalence checking between ISCAS'85 benchmarks and their non-redundant version show the robustness and the efficiency of our method. It takes only a minute to do the whole verification.

The rest of the paper is organized as follows. In section 2, we summarize previous works on combinational equivalence checking. In section 3, we describe our method and algorithm. Experimental results are shown in section 4. And section 5 is the conclusion.

2 Previous works

Binary Decision Diagrams(BDDs) based approach was proposed by Bryant[1]. This utilizes canonicality of BDDs to compare two logic functions. Fujita et al. and Malik et al. proposed heuristics for input variable ordering based on circuit structure[2, 3]. With these heuristics, they successfully make BDDs for some of the ISCAS'85 benchmark circuits' output function. Rudell proposed a more powerful ordering heuristic, called dynamic reordering[4]. Although this takes much time, *good* variable orderings for all the ISCAS'85 circuits except c6288 are derived. But these methods are not robust. Like c6288, there exist circuits whose BDD size grow exponentially to the number of inputs under any variable orderings. Therefore, a method that represents output functions using BDDs cannot be a core technique of combinational equivalence checking. Actually, the problem representing output functions using BDDs and equivalence checking problem is not equivalent. There are many circuits such that their output functions cannot be represent using BDDs efficiently, but equivalence checking of them is easily done.

Practically, there are some similarities between two circuits to be verified in many cases. So, if we can utilize such information, the complexity of the problem may be lowered. Berman and Trevillyan proposed such a framework of equivalence checking method[5]. For example, suppose two circuits A and B consist of two subcircuits A-1,A-2 and B-1,B-2, respectively. If we can verify that A-1 and B-1 are equivalent and A-2 and B-2 are also equivalent, we can conclude the entire circuits A and B are equivalent. But the inverse is not true. Consider two circuits in figure 1. For the input parts (the left hand of the dashed line), we can verify $s1$ and $s2$ are equivalent and $t1$ and $t2$ are equivalent. For the output parts (the right hand of the dashed line), since $x1 = s1 + t1$ and $x2 = s2 \oplus t2$, they are different. From these results, however, we can not state the two circuits are not equivalent. Actually, the entire circuits are equivalent, because $s1(s2)$ and $t1(t2)$ never be 1 at the same time.

Such a case is called *false negative*. In [5], Berman only points out the false negative problem, and no concrete procedure is shown.

Kunz proposed a powerful indirect implication method, called the *recursive learning*, and he applied it to the equivalence checking[6]. Reddy et al. combined

33rd Design Automation Conference ®

DAC 96 - 06/96 Las Vegas, NV, USA

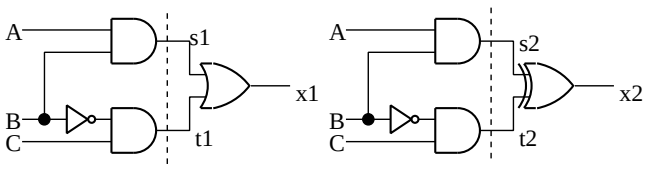

Figure 1: An example of the false negative

this recursive learning technique and BDD technique[7]. But their usage of BDDs is not effective. In the case of false negative, an ATPG based justification procedure is invoked. Generally, such an ATPG based technique is effective for problems that have solutions, but in the case of false negative, there is no solution, i.e. justification never succeeds.

Jain et al. proposed another indirect implication method, called the *functional learning*, and they also applied it to the equivalence checking[8]. The functional learning does similar implication as the recursive learning, but it uses BDDs to get implications between signal lines. If a proper cut line is given, the functional learning can be very effective. However, no heuristics on the cut line selection are given in their paper.

Furthermore, both of the methods of Reddy et al. and of Jain et al. only use the learning to get internal equivalent pairs. Because of the limitation of the learning (i.e. the limit of recursive level and the limit of cut line selection), an equivalent pair may not be found in their learning phase. Such loss of equivalence information may degrade the entire performance of equivalence checking, especially for large circuits. For example, suppose we try to verify two circuits in figure 2, and they consist of two parts, circuit1(circuit1') and circuit2. And suppose circuit1 and circuit1' are functionally equivalent but have different structures.

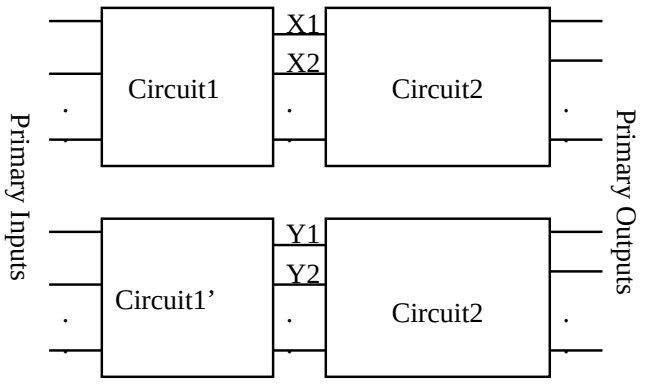

Figure 2: Verification of large circuits

If we can verify circuit1 and circuit1' are equivalent, we can easily verify the entire circuit, since the output parts of both circuits have the same circuit structure. However, if learning technique can not prove circuit1 and circuit1' are equivalent, no equivalent pairs are found in the output parts even though they have the same structure, and thus we have to try to verify the entire circuit. This seems more difficult than to try to verify circuit1 and circuit1'. One would say that with enough limit given, learning can detect all the equivalence pairs. But it is not practical. There is no method to automatically decide on what may be the sufficient for each learning. Too strict limit loses some equivalence information, and too loose limit takes much time and memory.

As shown in this section, potentially, there remains several points to improve the efficiency and the robustness of equivalence checking algorithm.

3 The algorithm of equivalence checking

3.1 Definitions

A **Boolean network** is a directed acyclic graph, $G = (V, E)$, representing a combinational circuit of technology independent level. A vertex v has a logic function F_v unless v is an primary input. A vertex u that corresponds to an input variable of F_v is called a **fanin** of v. Inversely, v is called a **fanout** of u in this case. A direct edge $u \rightarrow v$ exists only among such vertices u and v. We use FI_v to represent a set of all the fanins of v, and we use FO_v to represent a set of all the fanouts of v. Sets of vertices TFI_v and TFO_v defined in the following equations are called the **transitive fanins** and the **transitive fanouts**, respectively.

$$TFI_v = FI_v \cup (\bigcup_{u \in FI_v} TFI_u)$$
$$TFO_v = FO_v \cup (\bigcup_{u \in FO_v} TFO_u)$$

A logic function of vertex v with respect to primary inputs is called the **global function** of v, and is denoted as G_v. G_v is calculated as follows[1].

$$G_v = \exists u \in FI_v, F_v \wedge (\bigwedge_{u \in FI_v} (u \equiv G_u))$$

If the global function of v and the global function of u is equivalent, i.e. $G_v \equiv G_u$, we say v and u are **equivalent**. A pair of mutual equivalent vertices is called **equivalent pair**.

For a vertex v and a subset of its transitive fanins T, if any path from an primary input to v contains at least

[1] In this paper, a vertex and its corresponding variable are used interchangeably.

one element of T, such T is called a **basis** of v [2]. A logic function of v with respect to the vertices in T is called the **local function** of v over T, and is denoted as L_v^T. L_v^T is calculated in a similar way to the calculation of G_v. Obviously, for the set of all primary inputs PI, $L_v^{PI} \equiv G_v$ holds for any vertex v.

A network generated from two Boolean networks under checking with connecting corresponding primary inputs together is called a **composite network**(figure 3). The equivalence checking problem is a problem to check whether corresponding primary output pairs on a composite network are equivalent.

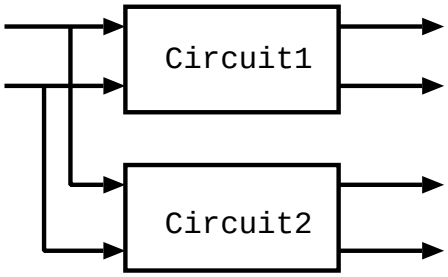

Figure 3: Composite network

3.2 Overview of the verification method

We use a general framework like Berman and Trevillyan's method[5](figure 4).

At first, random pattern simulation is done on the composite network, and the list of candidate equivalent pairs(**CEP-list**) is generated. Next, vertices pair (v_1, v_2) is extracted in breadth first order from primary inputs, and verification on that pair is invoked. If that pair is verified as equivalent, it is added to the list of verified equivalent pairs(**VEP list**) and reconnection about the equivalent pair is done. An example of the reconnection is shown in figure 5. In this case, vertex (s_1, s_2) and (t_1, t_2) are the verified equivalent pairs. All the fanouts of s_2 and t_2 are modified to connect to s_1 and t_1 instead of s_2 and t_2, and vertices that have no fanouts are deleted because they are useless. In this process, there is a choice which vertex is deleted. In our current implementation, further vertex from primary inputs is deleted.

Finally, we check primary outputs pairs are in the VEP-list.

3.3 Verification of equivalent pairs using BDDs

To verify a given pair of vertices is equivalent, we use a BDD based method. As described before, an approach that represents global functions using BDDs is not robust. Instead, we use local functions. If we can find a good basis that does not cause the false negative problem, equivalence checking is easily done by only calculating local functions of two vertices. Unfortunately, we can not always find a good basis, therefore, some mechanism is needed to handle the false negative problem.

[2] Exactly speaking, an element of a basis of v need not to be a member of the transitive fanins of v. But, such an element is useless for the local functions of v over T.

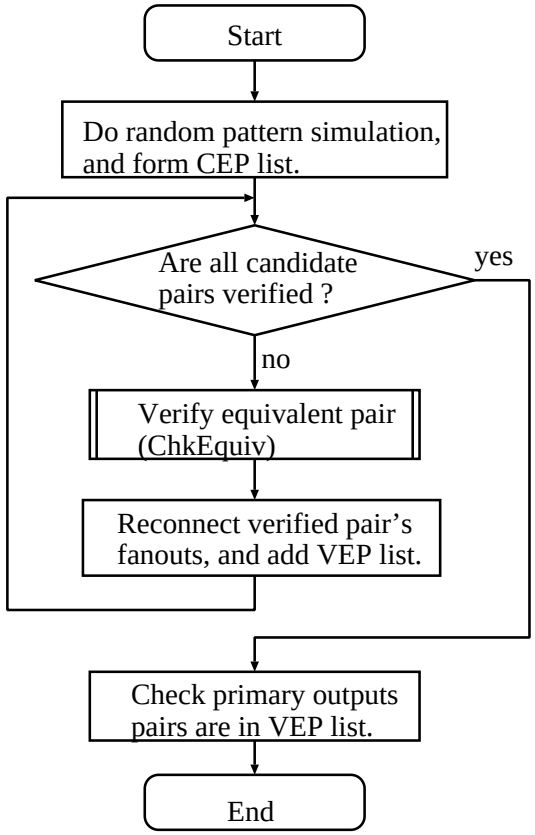

Figure 4: Flowchart of the verification method

For that purpose, we introduce a functional operation using BDDs, which is called the **functional implication**. As the name implies, this is similar to Jain's functional learning[8]. But its purpose and usage are different. In this paper, we define the **functional implication** as an operation that derives a combination of values at a set of vertices $T = \{t_1, t_2, \ldots, t_n\}$ in a Boolean network from a combination of values at another set of vertices $S = \{s_1, s_2, \ldots, s_m\}$. For example, consider the circuit in figure 6. Let $S = \{a, b\}$ and $T = \{c, d, e\}$. With the functional implication we get a combination of values at T, {(c=1,d=1,e=0),(c=0,d=1,e=1)}, from a combination of values at S, {(a=1,b=0),(a=0,b=1)}.

Kunz's recursive learning is a kind of constant value implication, and it does not represent a relation of more than two vertices like the above example. Jain's functional learning potentially do the same thing, however,

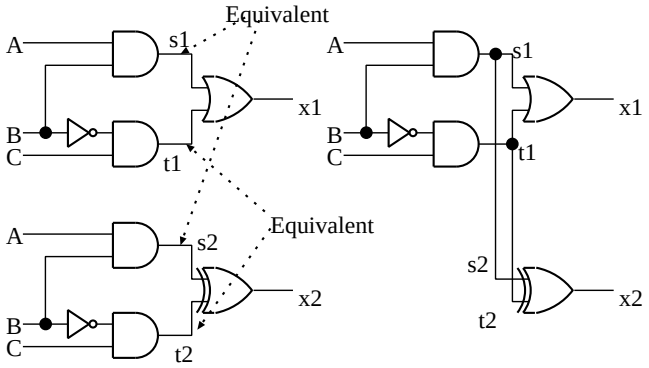

Figure 5: Reconnection

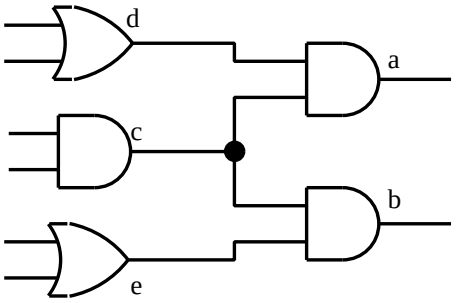

Figure 6: An example for functional implication

no concrete procedure is shown in the paper. We will describe how to choose S and T in 3.4.

The concept of the functional implication is not so novel. If we have the relation $\mathcal{R}$ between S and T, the functional implication can be done using the image computation. More concretely, say $F_S(S)$ is the characteristic function representing a combination of values on S, and $F_T(T)$ is the characteristic function representing corresponding combination of values on T. $F_T(T)$ is calculated as follows[3],

$$F_T(T) = \exists s \in S, (F_S(S) \wedge (\bigwedge_{s \in S} (s \equiv L_s^T)))$$

Recall that L_s^T is the local function of s over T. The above calculation is efficiently done by *compose* operation proposed by Bryant[1].

The verification algorithm using this functional implication is shown in figure 7.

At first, S is set to $\{v_1, v_2\}$ and $F_S(S)$ is set to $v_1 \oplus v_2$. Then we choose a set of vertices T, and do the functional implication. If v_1 and v_2 are equivalent and T is properly chosen, $F_T(T)$ becomes ϕ. Otherwise, there are two possibilities, v_1 and v_2 are not equivalent or the

[3] Through this paper, we assume any member of T is not contained in the transitive fanout of S.

```
boole ChkEquiv(v1, v2) {
    S ← {v1, v2};
    F_S ← v1 ⊕ v2;
    while (F_S ≠ φ) {
        if (S only consists of PIs) return FALSE;
        /* Derive the next vertices Set T from S.*/
        T ← GetT(S);
        /* the Do functional implication. */
        F_T ← FuncImp(F_S);
        S ← T;
        F_S ← F_T;
    }
    return TRUE;
}
```

Figure 7: The verification algorithm using functional implication

false negative occurs. To distinguish these two cases, T and $F_T(T)$ become new S and $F_S(S)$, and the new T is chosen, then the functional implication is invoked. This iteration continues until $F_S(S) = \phi$ or S becomes a set of primary inputs. If the former terminate condition holds, v_1 and v_2 are equivalent. Otherwise, v_1 and v_2 are not equivalent.

3.4 Finding good basis

The selection of basis T is essential for efficient computation of the algorithm in figure 7(*ChkEquiv*). If we choose a set of primary inputs as T, our algorithm is nothing but the global function based method like [2, 3]. To choose a proper set of vertices, we have the following observations.

Property-1 If there is no vertex v in T such that v is contained in more than one transitive fanins of vertices in S, there is no hope that $F_T(T)$ becomes ϕ [4].
⇒ A vertex that is contained in many transitive fanins of vertices in S should be selected.

Property-2 The false negative occurs if mutually dependent vertices are selected together into the same basis.
⇒ A set of mutually independent vertices are suitable for basis T.

From these consideration, we developed a heuristic to derive a proper basis T from S(figure 8).

In the first step, all the vertices in transitive fanins of S are labeled. For each vertex v, l_v records the number of vertices in S whose transitive fanins contain v. We want to select a vertex that has high l_v value according to Property-1. To do this, another value, denoted m_v is

[4] If there are vertices whose values are stuck, it is not the case.

```
set-of-vertices GetT(S) {
    foreach s ∈ S {
        foreach v ∈ TFI_s {
            l_v ++;
        }
    }
    foreach v {
        Calculate m_v.
    }
    do {
        Clear mark of all vertices.
        foreach s ∈ S {
            Search from s in depth first order,
            and mark a vertex v such that l_v = m_v.
        }
        chg ← FALSE;
        foreach v ∈ marked vertices {
            n ← the number of fanins without mark;
            if (n ≤ 1) {
                l_v ← 0;
                chg ← TRUE;
            }
        }
    } while (chg = TRUE);
    return a set of marked vertices;
}
```

Figure 8: The heuristic to derive a set of vertices

calculated as follows.

$$m_v = \max(l_v, (\max_{u \in FI_v} m_u))$$

Basically, a vertex v such that $m_v = l_v$ is a candidate for the basis. For any primary input v, $m_v = l_v$ always holds, therefore, a set of vertex v such that $m_v = l_v$ forms a basis.

Next, we adjust candidates according to Property-2. To exactly know that vertices are mutually independent, we need to construct the global functions of them. This is not practical, so we only use structural information for this adjustment. In general, if a vertex v is contained in the transitive fanins of a vertex u, u depends on v. Simply speaking, such a vertex u should be excluded from the candidates. But we may have to add compensating extra vertices instead of u to keep the candidates to be a basis. Because increase of vertices in T tends to drastically increase the size of BDD representing $F_T(T)$, we do not want to add extra vertices. In our current implementation, a vertex that has at most one compensating extra vertex is excluded. Exclusion of one vertex affects other vertices' exclude conditions. So, we continue this adjustment until stable state.

Figure 9 shows an example of derivation of basis T using this heuristic.

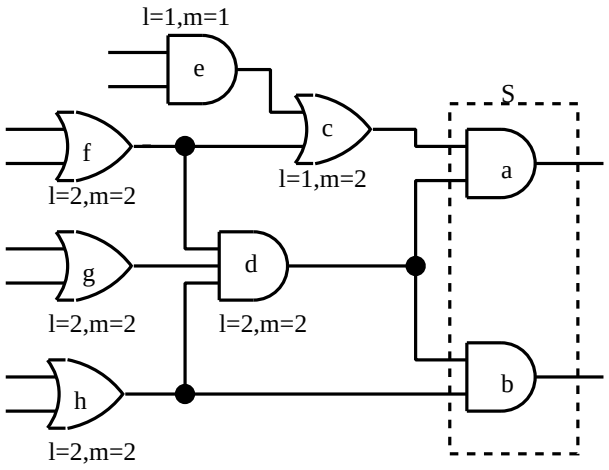

Figure 9: An example for basis selection

At first, the transitive fanins of a and b are labeled. In this example, all the vertices except c and e are contained in both transitive fanins, i.e. $l_c = l_e = 1, l_d = l_f = l_g = l_h = 2$. Then m_v is calculated. f is a fanin of c and $l_f = 2$, so, m_c becomes 2, not 1. For other vertices, m_v is equal to l_v. Next, candidates for T is selected. The first candidates are $\{e, f, d, h\}$. However, d has fanins f and h, which are also current candidates. If we exclude d from the candidates, we must include g as a compensating extra vertex. Since this exclusion does not increase the number of vertices in T, we really exclude d and add g to the candidates. The second candidates become $\{e, f, g, h\}$. There is no dependency among the current candidates, therefore, these candidates become the final result.

4 Experimental results

We have implemented the proposed algorithm in C++. The followings are the details of implementation.

- As the algorithm *ChkEquiv* is relatively slow, we try to verify an equivalent pair first using closest verified equivalent points as a basis. If two local functions over the basis are equivalent, further verification is not necessary. Otherwise, we invoke the algorithm *ChkEquiv*.
- Inversely equivalent pairs (e.g. $v_1 = \overline{v_2}$) are also considered as well as (true) equivalent pairs. Verification can be similarly done.
- Variable ordering is decided by the heuristic in figure 8 according to the order when marked.
- In the functional implication, vertices in S are sorted according to the mutual relation that is measured by the number of common basis vertices they have. Vertices are processed in this order. This aims at keeping the intermediate BDD small.

With this equivalence checker, we have some experiments. Table 1 shows the experimental results on SUN-4/10(90Mbytes memory) using ISCAS'85 benchmarks and their non-redundant versions (e.g. c432 vs. c432nr).

Table 1: Experimental results

name	no. of PIs	no. of POs	CPU(s)	
			[7]	Ours
c432	36	7	2.2	0.75
c499	41	32	2.17	1.23
c1355	41	32	6.73	3.37
c1908	33	25	14.54	6.21
c2670	233	139	159.3	3.93
c3540	50	22	67.64	17.38
c5315	178	123	372.8	13.96
c6288	32	32	32.74	9.12
c7552	207	108	5583.3	20.62

Similar experiments are done in [7], so we compare the results although they do not specify what kind of machine they used. Table 1 shows that our equivalence checker is quite faster than other existing methods especially, for c2670, c3540, c5315, and c7552, which are thought to be relatively difficult to verify. We also tried to verify the circuits in [8](FJ1 and FJ2). These two circuits are subcircuits of another circuit, so, we verify the whole circuit(i.e. FJ1 + FJ2 + α) in the similar manner to the above experiments. Our equivalence checker takes 23.4 seconds to verify the circuit, while in [8], the functional learning based method takes $590 + 23400$ seconds.

Through the above experiments 17244 candidate pairs were verified. For almost all pairs, the verification is trivial, i.e. the algorithm *ChkEquiv* is not invoked. Only for 206 pairs, the algorithm *ChkEquiv* is required. Table 2 shows internal statistics of such difficult verification.

Table 2: Internal statistics of the verification

	no. of pairs	no. of loops	
		Ave.	Max.
Success	188	1.27	5
Failure	18	2.60	5

The first column splits the pairs into to groups, pairs whose verification succeeded and pairs whose verification failed. The second column shows the number of such pairs. Rest of two columns are concerned with the number of loops in *ChkEquiv* — the former is the average number and the latter is the maximum number. This shows that in the case of successful verification, our basis selection heuristic in figure 8 works quite effective. About 75% of the pairs are verified with only one loop. Recall that *ChkEquiv* is invoked if a naive checking using closest equivalent points fails. In the case of failure, however, any basis selection heuristics does not work well, since $F_T(T)$ never be ϕ. Such non-equivalent pairs should be excluded by another heuristic.

5 Conclusion

In this paper, we propose an efficient and robust equivalence checking method. Our main contribution is the verification algorithm based on the functional implication and the heuristic to select a set of vertices used for a basis of the local functions. Experimental results show the efficiency of our method in practical field. Generally, BDD based algorithm like ours seems suitable for solving co-NP complete problem unless the size of BDDs blows up. On the other hand, verifying given vertices pair are not equivalent is a NP complete problem. For such a problem, only one input pattern that distingushes the two vertices is enough to prove the discrepancy. So, BDD based approach does not seem to be effective. In the experiments described in section 4, almost all non-equivalent pairs are filtered out by random pattern simulation. However, to make our equivalence checker more robust, interface with ATPG or structural based technique is necessary.

References

[1] R.E. Bryant, "Graph-based algorithms for boolean function manipulation", *IEEE Transactions on Computer*, C-35(12), 1986.

[2] M. Fujita, H. Fujisawa, and N. Kawato, "Evaluation and Improvements of Boolean Comparison Method Based on Binary Decision Diagrams", In *Proc. of ICCAD*, pp. 2-5, Nov. 1988.

[3] S. Malik, A. Wang, R. Brayton, and A. Sangiovanni-Vincentelli, "Logic Verification Using Binary Decision Diagrams in a Logic Synthesis Environment", In *Proc. of ICCAD*, pp. 6-9, Nov. 1988.

[4] R. Rudell, "Dynamic Variable Ordering for Ordered Binary Decision Diagrams", In *Proc. of ICCAD*, pp. 42-47, Nov. 1993.

[5] C.L. Berman and L.H. Trevillyan, "Functional Comparison of Logic Designs for VLSI Circuits", In *Proc. of ICCAD*, pp. 456-459, Nov. 1989.

[6] W. Kunz, "Hannibal: An Efficient Tool for Logic Verification Based on Recursive Learning", In *Proc. of ICCAD*, pp. 538-543, Nov. 1993.

[7] S.M. Reddy, W. Kunz, and D.K. Pradhan, "Novel Verification Framework Combining Structural and OBDD Methods in a Synthesis Environment", In *Proc. of 32nd DAC*, pp. 414-419, Jun. 1995.

[8] J. Jain, R. Mukherjee, and M. Fujita, "Advanced Verification Techniques Based on Learning", In *Proc. of 32nd DAC*, pp. 420-426, Jun. 1995.