Equivalence Checking Using Abstract BDDs

S. Jha[†] Y. Lu[‡] M. Minea[†] E. M. Clarke[†]

Computer Science Dept[†] Electrical and Computer Engineering Dept[‡]
Carnegie Mellon University, Pittsburgh, PA 15213
sjha, yuanlu, marius, emc@cs.cmu.edu*

Abstract

We introduce a new equivalence checking method based on abstract BDDs (aBDDs). The basic idea is the following: given an abstraction function, aBDDs reduce the size of BDDs by merging nodes that have the same abstract value. An aBDD has bounded size and can be constructed without constructing the original BDD. We show that this method of equivalence checking is always sound. It is complete for an important class of arithmetic circuits that includes integer multiplication. We also suggest heuristics for findings suitable abstraction functions based on the structure of the circuit. The efficiency of this technique is illustrated by experiments on ISCAS'85 benchmark circuits.

1 Introduction

Formal verification is becoming extremely important because the algorithmic complexity and therefore the size of VLSI circuits keeps increasing. Binary decision diagrams (BDDs) [1] have proved to be critical for the success of many of these verification techniques. BDDs can handle medium size circuits very efficiently. Numerous techniques have been developed in order to handle larger circuits [2, 5, 7].

Although these methods are useful, none provides an upper bound for the BDD size. Thus, the node explosion problem for BDDs still exists. The Chinese remainder theorem was used by Clarke, Grumberg and Long in [3] to reduce the BDD size in the verification of a sequential multiplier. Kimura [4] has extended this idea and introduced residue BDDs, which have bounded size. He has used them successfully to verify a 16×16 combinational multiplier. In [6] Ravi *et al.* discuss how to choose a good modulus and also show how to build the residue BDD for complex circuits involving function blocks.

In this paper, we generalize the idea of residue BDDs and define a new data structure called an *abstract BDD* (aBDD). Given an abstraction function, aBDDs reduce the size of BDDs by merging nodes that have the same abstract value. aBDDs have bounded size and can be built directly from combinational circuits. Residue BDDs are a special case of aBDDs. In fact, our results explain exactly when residue BDD techniques work.

There are several advantages of aBDDs over residue BDDs. Residue BDDs are particularly well suited for arithmetic circuits, but do not work well for control circuits. As our experimental results show, the use of other abstraction functions leads to significantly better results for control circuits. In addition, aBDDs provide designers the flexibility to choose abstraction functions based on the nature of the circuits, for example by taking into account symmetry. This paper demonstrates that aBDDs can be used as a general framework for applying abstraction to BDDs.

2 Abstract BDDs

Let $B = \{0, 1\}$. B^n is the set of 0-1 vectors of size n. A 0-1 vector of size i will be denoted by $\vec{x} = (x_0, \cdots, x_{i-1})$. The concatenation of vectors $\vec{x}$ and $\vec{y}$ is written as $\vec{x} \cdot \vec{y}$. For example, $(0, 0, 1) \cdot (1, 0)$ is the vector $(0, 0, 1, 1, 0)$. The symbol $\vec{0}_i$ represents the vector of all zeroes of length i.

An *abstraction function* is a function $\alpha : B^n \rightarrow D$, where D is some arbitrary range. In general, an abstraction function may map multiple values of the domain B^n to a single value in the range D. Usually the range D will be much smaller than the domain B^n. The size of D is denoted by $|D|$.

An abstraction function $\alpha^i : B^i \rightarrow D$ induces an equivalence relation on vectors in B^i: for $\vec{x}, \vec{y} \in B^i$ define $x \cong y$

*This research was sponsored in part by the National Science Foundation under grant no. CCR-9217549, by the Semiconductor Research Corporation under contract 96-DJ-294, and by the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and the Advanced Research Projects Agency (ARPA) under grant F33615-93-1-1330.

iff $\alpha^i(x) = \alpha^i(y)$. The equivalence relation $\cong$ partitions the 0-1 vectors into equivalence classes. We choose a unique representative from each equivalence class and construct a representative function h^i such that $h^i(x)$ is the unique representative in the equivalence class of x. From the initial abstraction function α we have thus generated a function $h^i : B^i \to B^i$. If n is the length of $\vec{x}$, then we write $h(\vec{x})$ to denote $h^n(\vec{x})$. It is easy to see that h is idempotent, i.e, $h(h(\vec{x})) = h(\vec{x})$. Next, we define what it means for the abstraction function function h to be *consistent*.

Definition 1 An abstraction function $h : \bigcup_j^n B^j \to \bigcup_j^n B^j$ is *consistent* iff for all $1 \leq i \leq n$, for all $\vec{x}, \vec{y} \in B^i$ the following implication is true.

$$h(\vec{x}) = h(\vec{y}) \quad \Rightarrow \quad \forall \vec{z}\left[h(\vec{x} \cdot \vec{z}) = h(\vec{y} \cdot \vec{z})\right]$$

For example, consider the abstraction function $h : \bigcup_j^n B^j \to \bigcup_j^n B^j$ induced by a linear abstraction function $\alpha^i, i = 0, \cdots, n-1$ defined below:

$$\alpha^i(x_0, \cdots, x_i) \quad = \quad \sum_{j=0}^{i} b_j x_j$$

where b_j $(0 \leq j \leq i)$ are real numbers. It is not hard to see that h is a consistent abstraction function. As a different example, the function that computes the residue of a positive integer with respect to a prime number is also a consistent abstraction function (we assume the usual conversion between integers and bit vectors). The consistency property will permit the same abstraction mapping to be applied at different levels in a BDD. In the remainder of this paper we will assume that all abstraction functions are both consistent and idempotent.

Let $f : B^n \to B$ be an n-argument boolean function. An abstraction function $h : B^n \to B^n$ induces a transformation on boolean functions according to the following relation. We denote the transformed function as f_h and define it as follows:

$$f_h \quad = \quad f \circ h$$

Lemma 1 *Let $f, p, q : B^n \to B$ be boolean functions, $\odot$ any logical operation, and $h : B^n \to B^n$ be an abstraction function. If $f = p \odot q$, then $f_h = p_h \odot q_h$.*

Proof: Let $\vec{x} \in B^n$ be an arbitrary vector. We have the following equations:

$$\begin{aligned} f_h(\vec{x}) &= (f \circ h)(\vec{x}) \\ &= f(h(\vec{x})) \\ &= p(h(\vec{x})) \odot q(h(\vec{x})) \\ &= p_h(\vec{x}) \odot q_h(\vec{x}) \end{aligned}$$

The result follows. □

Next, we show how the above results can be applied when representing boolean functions by binary decision graphs.

Definition 2 A *levelized binary decision graph* (levelized BDG) with n levels is a 7-tuple $(V, left, right, level, t_0, t_1, root)$, where

- V is the set of nodes.
- $level : V \to \{0, \cdots, n\}$.
- $left : (V\text{-}\{t_0, t_1\}) \to V$ is the *left child* function with restriction: $level(v) = level(left(v)) - 1$.
- $right : (V\text{-}\{t_0, t_1\}) \to V$ is the *right child* function with restriction: $level(v) = level(right(v)) - 1$.
- $t_0 \in V$ is the *zero* node with $level(v) = n$.
- $t_1 \in V$ is the *one* node with $level(v) = n$.
- $root \in V$ is the distinguished root node and $level(root) = 0$.
- For all $v \in V\text{-}\{root, t_0, t_1\}$, $1 \leq level(v) \leq n-1$.

Given a levelized BDG T, we define a function $node_T : \bigcup_{i=1}^n B^i \to V$. Let $p \in B^i$ be a 0-1 vector or path of length i. $node_T(p) = v$ iff we get to node v by following the path p from the root. Notice that a levelized BDG T corresponds to a boolean function $b(T) : B^n \to B$ in the following manner:

- $b(T)((y_1, \cdots, y_n)) = 0$ iff $node_T((y_1, \cdots, y_n)) = t_0$.
- $b(T)((y_1, \cdots, y_n)) = 1$ iff $node_T((y_1, \cdots, y_n)) = t_1$.

Given an abstraction function $h : B^n \to D$, we show how to construct an *abstract* levelized BDG from a given levelized BDG. Without loss of generality, assume we have chosen the representative x of an equivalence class to be the lexicographically least element in that equivalence class. Therefore, if $\leq$ denotes lexicographical ordering, and $h(x) = x$, then $x \leq y$ for all y such that $h(x) = h(y)$.

The algorithm that constructs an abstract levelized BDG is given in Figure 1. Its arguments are a node $v \in V$ and a vector $path \in \cup_{i=0}^n B^i$ representing the path from the root to that node. The initial call to the algorithm is $DFS(root, \epsilon)$, where ϵ denotes the empty vector. The algorithm maintains a *cache* of pairs $(v, path)$, which is initially empty. The routine $lookup\_cache(p')$ returns the node v' such that (v', p') is in the cache.

Lemma 2 Let T be a levelized BDG, $h : B^n \to B^n$ be an abstraction function, and T_h be the corresponding abstract levelized BDG as constructed by the *DFS* algorithm. Then the boolean function $b(T_h)$ corresponding to T_h is the transformation under h of the boolean function $b(T)$ corresponding to T:

$$\begin{aligned} b(T_h) &= b(T)_h \\ &= b(T) \circ h \end{aligned}$$

```
function DFS(v, path)
p' = h(path);
if p' ≠ path
    v' = lookup_cache(p');
    return v';
else
    if nonterminal(v)
        left(v) = DFS(left(v), path · (0));
        right(v) = DFS(right(v), path · (1));
    endif;
    insert_cache(p', v);
    return v;
endif
```

Figure 1. Abstraction of levelized BDG

Proof: We prove this lemma by induction on the length of path p. First, the root is considered when $p = \epsilon$. it is trivial to see that $node_{T_h}(p) = node_T(h(p)) = root$. Let us assume that $node_{T_h}(p) = node_T(h(p))$ is true for $length(p) = i$. Then consider a path $p \cdot y$ where $y \in B$. It is easy to see that both $node_{T_h}(p \cdot y)$ and $node_T(h(p \cdot y))$ are at level $i + 1$. There are two cases:

- $p \cdot y = h(p \cdot y)$, from program in Figure 1, we know that
$$node_{T_h}(p \cdot y) = node_T(h(p \cdot y))$$
- $p \cdot y \neq h(p \cdot y)$, from induction hypothesis, we have
$$node_{T_h}(p) = node_T(h(p))$$
Furthermore according to program in Figure 1,
$$node_{T_h}(p \cdot y) = node_T(h(h(p) \cdot y))$$
Since h is idempotent, $h(p) = h(h(p))$ implies $h(p \cdot y) = h(h(p) \cdot y)$ according to the definition of consistent function. Thus, we have
$$node_{T_h}(p \cdot y) = node_T(h(p \cdot y)).$$
By induction, this is true for the last level, which implies
$$b(T_h) = b(T) \circ h$$

□

Given a levelized BDG T, let n_i be the number of nodes $v \in V$ whose level is i. The *width* of T is $\max_{i=0}^{n-1} n_i$.

Lemma 3 *Given an abstraction function $h : B^n \to D$ and a levelized BDG T, the width of T_h is less than or equal to $|D|$.*

Proof: Let p_1 and p_2 be two paths of length i such that $h(p_1) = h(p_2)$. In the levelized BDG T_h, we have $node_{T_h}(p_1) = node_{T_h}(p_2)$. Thus, if two paths p_1 and p_2 agree on the abstraction value, then they lead to the same node. Hence, at each level the number of nodes in the levelized BDG T_h is bounded by the size of the range of h. □

Let $\odot$ be an arbitrary operation on boolean functions. The lemma given below states that abstraction of levelized BDGs can be performed compositionally.

Lemma 4 *Assume that we have three levelized BDGs T, T^1, and T^2. If $b(T) = b(T^1) \odot b(T^2)$, then we have the following equation:*

$$b(T_h) \quad = \quad b(T_h^1) \odot b(T_h^2)$$

Proof: The proof follows from the following equations:

$$\begin{aligned} b(T_h) &= b(T)_h \text{ (By Lemma 2)} \\ &= b(T^1)_h \odot b(T^2)_h \text{ (By Lemma 1)} \\ &= b(T_h^1) \odot b(T_h^2) \text{ (By Lemma 2)} \end{aligned}$$

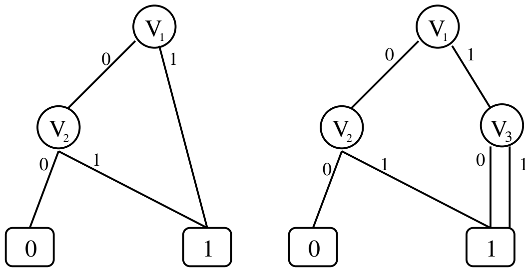

Figure 2. BDD and levelized BDD for $(x_0 \vee x_1)$

Levelized BDDs are obtained from levelized BDGs in the following manner: Given a levelized BDG T, we merge two nodes v and v' (whose level is the same) iff the subtrees rooted at them are isomorphic. Reduced ordered BDDs, on the other hand, add an extra level of optimization because redundant nodes are removed, as described in [1]. For example, Figure 2 gives the BDD for the function $(x_0 \vee x_1)$ and the corresponding levelized BDD. Because of the merging of isomorphic subtrees, we must modify algorithm DFS. We call our new algorithm BDD_DFS and it is described in Figure 3. In the algorithm given in Figure 3, $Sub(v)$ denotes the subtree rooted at the node v. $Sub(v) \approx Sub(v')$ means that the trees rooted at v and v' are isomorphic.

Given a levelized BDD T, the BDD T_h obtained from algorithm *BDD_DFS* is called an abstract BDD or aBDD. Notice that the aBDD obtained in this manner is also levelized. The definition of aBDDs and the properties proved in this chapter can be easily generalized for different abstraction functions used at each level.

```
function BDD_DFS(v, path)
p' = h(path);
if p' ≠ path
    v' = lookup_cache(p');
    return v';
else
    if nonterminal(v)
        left(v) = DFS(left(v), path · (0));
        right(v) = DFS(right(v), path · (1));
        if there exists v_1 in cache such that Sub(v) ≈ Sub(v_1)
            return v_1;
        endif;
    endif;
    insert_cache(p', v);
    return v;
endif
```

Figure 3. Modified DFS of levelized BDDs

3 Uniqueness of Representation

Assume that we have two boolean functions f and g and let T_f and T_g be the levelized BDDs for f and g. Given an abstraction function h, $h(T_f) \neq h(T_g)$ implies that $f \neq g$, but $h(T_f) = h(T_g)$ does not necessarily imply that $f = g$. In other words, aBDDs are not canonical. In this section we prove that with some restrictions we can obtain the canonicity property for a group of functions.

A set of abstraction functions $\{h_1, \cdots, h_p\}$, where $h_i : B^n \to B^n$ for $1 \leq i \leq p$ is said to *preserve* the domain B^n iff given two vectors $\vec{x}$ and $\vec{y}$ such that $\vec{x} \neq \vec{y}$ there exists a k such that $h_k(\vec{x}) \neq h_k(\vec{y})$. As an example, for $n = 32$, the abstraction functions corresponding to taking the modulus with respect to 2, 3, 5, 7, 11, 13, 17, 19, 23, 29 preserve the domain $\{0, 1\}^{32}$. This follows from the Chinese remainder theorem.

Let $f : B^n \to B^n$ be a function and $h : B^n \to B^n$ be an abstraction function. Given a function $f : B^n \to B^n$, we represent it by a vector of n boolean functions $(f_1, \cdots, f_n)$. Assume that we are given a function f and a set of p abstraction functions $\{h_1, \cdots, h_p\}$ where $h_i : B^n \to B^n$. Let $(f_1, \cdots, f_n)$ be the array of boolean functions corresponding to f. Let $T^{i,j}$ be the levelized BDD corresponding to the boolean function $f_i \circ h_j$. Let $M(f)$ be a $m \times p$ matrix of aBDDs such that $M(f)_{i,j} = T^{i,j}$. The matrix is schematically shown below:

$$\begin{bmatrix} T^{1,1} & \cdots & T^{1,p} \\ T^{2,1} & \cdots & T^{2,p} \\ \cdots & \cdots & \cdots \\ T^{m,1} & \cdots & T^{m,p} \end{bmatrix}$$

The theorem given below proves that under certain conditions $M(f)$ is a canonical representation for f.

Theorem 1 Assume that $f : B^n \to B^n$ and $g : B^n \to B^n$ are two functions. Let $(f_1, \cdots, f_n)$ and $(g_1, \cdots, g_n)$ be the corresponding arrays of boolean functions. Also assume that $h_i : B^n \to B^n$ $(1 \leq i \leq p)$ is a set of abstraction functions that preserves the domain B^n. If we have $h_i \circ f = h_i \circ f \circ h_i$ and $h_i \circ g = h_i \circ g \circ h_i$ for all $1 \leq i \leq p$, then $f = g$ iff $M(f) = M(g)$.

Proof

It is obvious that if $f = g$, the corresponding matrices are equal, $M(f) = M(g)$. Consider the case $f \neq g$. This means that there exists a vector $\vec{a} \in B^n$ such that $f(\vec{a}) \neq g(\vec{a})$. Since the set of abstraction functions h_i preserves the domain B^n, there exists a k, where $h_k(f(\vec{a})) \neq h_k(g(\vec{a}))$. From the hypothesis, we conclude that $h_k(f(h_k(\vec{a}))) \neq h_k(g(h_k(\vec{a})))$, and therefore, $f(h_k(\vec{a})) \neq g(h_k(\vec{a}))$. Since both f and g are arrays of boolean functions, there must be a j for which $f_j(h_k(\vec{a})) \neq g_j(h_k(\vec{a}))$. This means that $f_j \circ h_k$ is different from $g_j \circ h_k$ and therefore $M(f) \neq M(g)$. $\square$

As an example, consider the function $mult : B^n \to B^n$ which multiples two integers with $\frac{n}{2}$ bits (we are assuming no overflow). For a vector $\vec{x} = (x_0, \ldots, x_{i-1}) \in B^i$ we define $val(\vec{x}) = \sum_{j=0}^{i-1} x_j * 2^j$. The function $mult$ is defined by the following equation:

$$val(mult(\vec{x})) \quad = \quad val(x_0, \cdots, x_{\frac{n}{2}-1}) * val(x_{\frac{n}{2}}, \cdots, x_n)$$

Assume that we have m relatively prime positive integers $p_1, \ldots, p_m$ such that $p_1 \cdot p_2 \cdots p_m \geq 2^n$. Let $h_i : B^n \to B^n$ be the abstraction function corresponding to taking the residue with respect to p_i. By the Chinese remainder theorem, the set of abstraction functions $\{h_1, \cdots, h_m\}$ preserves the domain B^n. Moreover, $h_i \circ f = h_i \circ f \circ h_i$ because

$$(x * y) \bmod p_i \quad = \quad ((x \bmod p_i) * (y \bmod p_i)) \bmod p_i$$

for any positive integer p ($*$ denotes the multiplication of integers). Translated into our notation the equation given above becomes

$$h_i \circ mult \quad = \quad h_i \circ mult \circ h_i$$

Therefore, $mult$ satisfies the condition in the hypothesis and the theorem applies to this case. More generally, this theorem will be also true when f satisfies $h_i \circ f = f \circ h_i$.

4 Equivalence checking using aBDDs

Because of their bounded size, aBDDs can be used to verify the equivalence of large circuits. In general, residue functions are good abstractions for arithmetic circuits. Symmetric and linear functions tend to work better for control logic. If the circuit has symmetric inputs, a symmetric abstraction function should definitely be used. These conclusions are supported by the experimental data in section 5. The overall procedure is as follows.

1. Given a circuit, choose a set of appropriate abstraction functions.

2. Select an abstraction function h out of a set of abstraction functions. This set will be provided based on the nature of the circuit.

3. Build aBDDs for the specification and the implementation circuit using the abstraction function h.

4. Compare the two aBDDs that are obtained for specification and implementation. If they are different, an error is detected. Otherwise, choose a different abstraction function from the set and repeat step 3 with a different abstraction function.

In general, there is no procedure to select a set of abstraction functions that will detect all errors in a circuit. Nevertheless, we believe that our methodology can be extremely useful in practice, since an initial design is much more likely to contain errors than to be correct.

Next, we give a description of our algorithm. Since our algorithm to build an aBDD assumes that we are working with a levelized BDD, we have to levelize a BDD before we apply our abstraction algorithm. For example, assume that $f = p \wedge q$ and that we have already built the aBDDs for p and q (with respect to the abstraction function h). Let us call these aBDDs $h(T_p)$ and $h(T_q)$. Next, we build the BDD corresponding to $h(T_p) \wedge h(T_q)$. Finally we levelize the BDD and apply our abstraction algorithm to obtain the aBDD for f.

We use a simple example to illustrate the algorithm. Assume that we have an abstraction function h for the circuit in Figure 4. The aBDD associated with z is $h(T_z)$. At the beginning, assume that we have aBDDs for the inputs a, b, c, d. By performing the $nand$ operation, we form the BDD $T_e = \neg[h(T_a) \wedge h(T_b)]$. Next we perform abstraction on the levelized BDD of T_e and obtain the aBDD $h(T_e)$. The same procedure is performed on f. After we obtain aBDDs for e and f, we compute the aBDD for the output g by using the same method.

5 Experimental Results

We have implemented our algorithm in C. Our experiments were performed on a Sun SPARC 10 workstation with 200 Mbytes of memory. The experiments were performed on the ISCAS'85 benchmark circuits. Faults in the circuit were injected one by one by selecting a stuck-at fault on one input of an arbitrary gate. Table 1 compares our method with ordinary BDD equivalence checking. In the table, two abstraction functions are used. One is the symmetric abstraction function

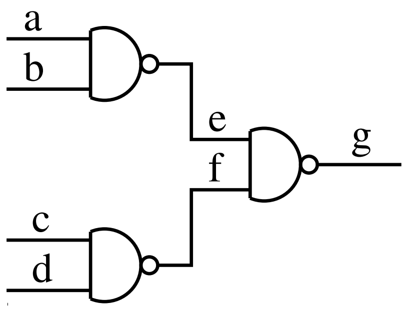

Figure 4. Building an aBDD from a circuit

$$\alpha^i(x_0, \cdots, x_i) = \sum_{j=0}^{i} x_j, i = 0, \cdots n-1$$

and the second one is the residue function

$$\alpha^i(x_0, \cdots, x_i) = \sum_{j=0}^{i} 2^j x_j \bmod n, i = 0, \cdots n-1$$

In Table 1, *Det Errs* is the number of faults detected by these three methods, and *Max # Nodes* is the maximum number of BDD nodes that need to be held in memory, which is usually much larger than the final BDD size. *Avg.Time* is the average time to detect a design error. The OBDD results for c2670, c5315, c6288 and c7552 are not reported because they exceeded the memory limit.

The experimental results show that using aBDDs it is possible to detect a high percentage of design errors (between 40% and 90% of the errors are detected using symmetric abstraction functions alone. The reduction in BDD size for some circuits is over two orders of magnitude. Since the size of aBDDs is bounded, this reduction will be much more significant in real industrial circuits. For most of the circuits, residue abstraction functions do not detect as many errors as symmetric abstraction functions. We choose the number of variables as a modulus because both symmetric and residue abstraction functions produce BDDs of similar size. The results support our argument that residue functions may not be a good abstraction for control circuits.

6 Conclusion

In this paper, we introduce a general framework for applying abstraction to BDDs. Abstract BDDs (aBDDs) are of bounded size and can be constructed directly from the circuit, without first generating the original BDDs. This technique makes it possible to show inequivalence of combinational circuits. If the aBDDs for two circuits are different, then the circuits correspond to two different boolean functions. On the other hand, if the two aBDDs are identical, the

circuits	Errs	Det Errs			Max # Nodes			Avg.Time		
		OBDD	Symm	Resid	OBDD	Symm	Resid	OBDD	Symm	Resid
c432	50	50	50	33	4712	4604	3902	1.15	7.94	19.70
c499	50	50	40	28	95745	9481	27121	22.74	16.72	48.64
c880	50	50	28	7	637338	7705	4999	138.25	58.17	180.56
c1355	50	50	40	28	96357	9497	27476	25.49	44.93	129.48
c1908	50	48	40	36	70196	6274	15838	35.95	22.82	61.86
c2670	10	unable	5	2	–	132593	774009	–	5449.37	5073.17
c3540	50	50	24	16	1522988	9927	8267	299.89	109.61	379.06
c5315	10	unable	10	3	–	208795	234716	–	4618.01	10052.3
c6288	10	unable	6	6	–	7317	38	–	86.52	61.20
c7552	10	unable	9	10	–	366462	2301523	–	11963.65	18405.8

Table 1. Comparison of equivalence checking using OBDDs and aBDDs

circuits may still be different. In spite of this lack of completess, experimental results show that the technique is able to find a suprisingly large number of errors in practice. This is important because circuits tend to contain errors much more frequently than they are correct. Moreover, we identify an important class of functions for which this technique is complete. This class includes many common arithmetic circuits including integer multiplication. We are currently investigating probabilistic techniques for estimating the error coverage obtained using this method.

References

[1] Randal E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation", *IEEE Trans. on Comput.*, Vol. C-35, No.8, pp.677-691, Aug. 1986.

[2] Karl S. Brace, Richard L. Rudell, Randal E. Bryant, "Efficient Implementation of a BDD Package", *27th Design Automation Conference*, pp. 40-45, 1990.

[3] Edmund M. Clarke, Orna Grumberg, David E. Long, "Model Checking and Abstraction", *ACM Transactions on Programming Languages and System*, Vol.16, No.5, pp.1512-1542, Sept. 1994.

[4] Shinji Kimura, "Residue BDD and Its Application to the Verification of Arithmetic Circuits", *32nd Design Automation Conference*, 1995.

[5] Hiroyuki Ochi, Koichi Yasuoka, Shuzo Yajima, "Breadth-First Manipulation of Very Large Binary-Decision Diagrams", *Proc. Intl. Conf. Comput. Aided Design*, pp.48-55, 1993.

[6] Kavita Ravi, Abelardo Pardo, Gary D. Hachtel, Fabio Somenzi, "Modular Verification of Multipliers", *Formal Methods in Computer-Aided Design*, pp.49-63, Nov. 1996.

[7] Richard Rudell, "Dynamic Variable Ordering for Ordered Binary Decision Diagrams", *Proc. Intl. Conf. Comput. Aided Design*, pp.42-47, 1993.