Equivalence Checking Using Abstract BDDs

S. Jha Y. Lu M. Minea E. M. Clarke

October 29, 1996
CMU-CS-96-187

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890

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 complete for an important class of arithmetic circuits that includes integer multiplication. The efficiency of this technique is illustrated by experiments on ISCAS'85 benchmark circuits.

The reseach was sponsored in part by the National Science Foundation (NSF) under grant no. CCR-9217549, by the Semiconductor Research Corporation (SRC) under contract 95-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.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of NSF, SRC, ARPA or the U.S. Government.

Keywords: consistent function, binary decision graph, binary decision diagram(BDD), abstract BDD.

1 Introduction

Formal verification is becoming extremely important as 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. Some of these techniques are improved data structures, typed edges, static variable ordering, and dynamic variable reordering. Breadth-first techniques have been developed that reduce cache misses during traversal. Recently, a new data structure called a Binary Moment Diagram (BMD) [3] has been introduced to deal with arithmetic circuits. Hybrid Decision Diagrams (HDDs) [6] combine MTBDDs [4] and BMDs to deal with both control and arithmetic circuits.

Although all of these methods are useful, none provides an upper bound for the BDD size. Thus, the node explosion problem for BDDs still exists. Kimura [7] has proposed the use of residue BDDs, which have bounded size, to overcome this problem. He has used them successfully to verify some large arithmetic circuits. 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. We prove that 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. Some of the disadvantages of residue BDDs are also eliminated. For example, it is known that residue BDDs may not work well on control circuits. Our results show that aBDDs can be used effectively for both arithmetic and control circuits, providing in some cases a size reduction of more than two orders of magnitude.

Our paper is organized as follows. In Section 2 contains the basic definitions, lemmas and theorems that are needed to insure the correctness of our method. A condition that guarantees a canonical representation by aBDDs is given in Section 3. Section 4 discusses a method for equivalence checking using aBDDs. Some experimental results are reported in in Section 5, and we conclude with Section 6.

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 surjection $a : B^n \rightarrow D$, 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 $a : B^n \rightarrow D$ induces an equivalence relation on vectors in B^n: for $\vec{x}, \vec{y} \in B^n$ define $x \cong y$ iff $h(x) = h(y)$. The equivalence relation $\cong$ partitions the 0-1 vectors into equivalence classes. We choose unique representatives from each equivalence class and construct a representative function h such that $h(x)$ is the unique representative in the equivalence class of x. From the initial abstraction function a we have thus generated a function $h : B^n \rightarrow B^n$. Hence, we can assume without loss of generality that the range of the abstraction function is B^n. Moreover, it is easy to see by construction that h is idempotent, i.e, $h(h(\vec{x})) = h(\vec{x})$. Next, we define what it means for the abstraction function function $h : B^n \rightarrow B^n$ to be *consistent*.

Definition 1 An abstraction function $h : B^n \rightarrow B^n$ is called *consistent* iff for all $1 \leq i \leq n$, $\forall \vec{x}, \vec{y} \in B^i$ the following equation is true.

$$h(\vec{x} \cdot \vec{0}_{n-i}) = h(\vec{y} \cdot \vec{0}_{n-i}) \quad \Rightarrow \quad \forall \vec{z} \in B^{n-i} \ [h(\vec{x} \cdot \vec{z}) = h(\vec{y} \cdot \vec{z})]$$

For example, consider the abstraction function $h : B^n \rightarrow B^n$ induced by a linear abstraction function a defined below:

$$a((x_0, \cdots, x_{n-1})) \quad = \quad \sum_{i=0}^{n-1} b_i x_i$$

where b_i $(0 \leq i \leq n-1)$ 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). In the remainder of this paper we will assume that all abstraction functions are both consistent and idempotent.

An abstraction function $h : B^n \rightarrow B^n$ induces an abstraction function $h^i : B^i \rightarrow B^i$ in the following way: given an $\vec{x} \in B^i$, $h^i(\vec{x})$ is the vector consisting of the first i bits of $h(\vec{x} \cdot \vec{0}_{n-i})$. When the size of the domain is clear from the context, we will still use h to denote h^i.

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

$$f_h \;=\; f \circ h$$

Lemma 1 *Let $f, p, q : B^n \rightarrow B$ be boolean functions, $\odot$ any logic operation, and $h : B^n \rightarrow B^n$ 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. □

We next 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.
- $left : (V\text{-}\{t_0, t_1\}) \rightarrow V$ is the *left child* function with the restriction that $level(v) = level(left(v)) - 1$.
- $right : (V\text{-}\{t_0, t_1\}) \rightarrow V$ is the *right child* function with the restriction that $level(v) = level(right(v)) - 1$.
- $level : V \rightarrow \{0, \cdots, n\}$.

- $t_0 \in V$ 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 $v \in V\text{-}\{t_0, t_1\}$, $1 \leq level(v) \leq n-1$.

Given a levelized BDG T, we define a function $node_T : \bigcup_{i=1}^{n} B^i \rightarrow V$. 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 $b(T) : B^n \rightarrow B$ 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 $h : B^n \rightarrow D$, 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, $h : B^n \rightarrow B^n$ 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: Pseudocode for transformation of levelized BDG

Proof: Given a path p, the arguments to our procedure are $(node_T(p), p)$. The algorithm *DFS* visits nodes in lexicographical order of their paths. Let p be a path such that $h(p) = p$. Since p is lexicographically less than all the paths p' such that $h(p') = p$, the node $node_T(p)$ will be visited before all the nodes $node_T(p')$ where $h(p') = p$. Hence, when the procedure is invoked with the parameters $(node_T(p'), p')$ the pair $(node_T(h(p')), h(p'))$ is already in the cache. In this case, *DFS* will return the node $node_T(h(p'))$, and hence $node_{T_h}(p') = node_T(h(p'))$. We still must prove that this relation is true for any node reached in the subtree rooted at $node_T(p')$ by following down an arbitrary path y. Since the subtree of $node_{T_h}(p')$ is same as the subtree of $node_{T_h}(p)$, we have $node_{T_h}(p' \cdot y) = node_{T_h}(p \cdot y) = node_T(h(p \cdot y)) = node_T(h(p' \cdot y))$. The last equality holds because h is a consistent abstraction function and $h(p') = h(p)$, and therefore $h(p' \cdot y) = h(p \cdot y)$. Since we have now proved that $node_{T_h}(p) = node_T(h(p))$ holds for any path p, the desired result follows from the definition of $b(T_h)$. □

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 done 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:

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

Levelized BDDs are obtained from levelized BDGs in the following manner: Given an 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 $(x_0 \vee x_1)$. Figure 3 shows the corresponding levelized BDD for the function $(x_0 \vee x_1)$. 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 4. Given a levelized BDD T (which is a levelized BDG), T_h is called an abstract BDD or aBDD. Notice that the aBDD obtained in this manner is levelized.

In the algorithm given in Figure 4, T_v denotes the subtree rooted at the node v. $T_v \approx T_{v'}$ means that the trees rooted at v and v' are isomorphic.

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, $h(T_f) \neq h(T_g)$

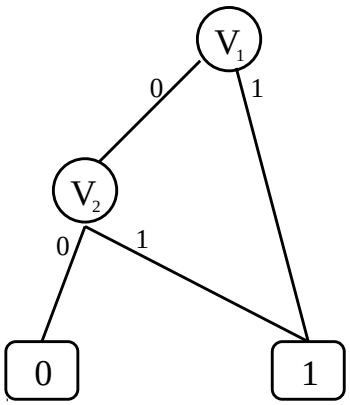

Figure 2: Example:BDD for $x_0 \vee x_1$

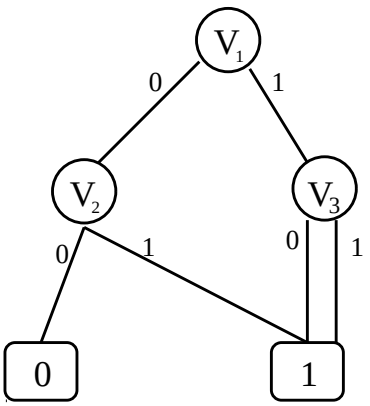

Figure 3: Example:Levelized BDD for $(x_0 \vee x_1)$

```
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 T_v ≈ T_{v_1}
            return v_1;
        endif;
    endif;
    insert_cache(p', v);
    return v;
endif
```

Figure 4: Pseudocode for modified DFS of levelized BDDs

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.

Definition 3 A function $f : B^n \rightarrow B^n$ *respects* the abstraction function $h : B^n \rightarrow B^n$, if and only if

$$\forall x, y \ . \ h(x) = h(y) \Rightarrow f(x) = f(y)$$

In particular, $f(h(x))$ respects h for any abstraction function $h : B^n \rightarrow B^n$.

Lemma 5 *Let $f : B^n \rightarrow B$ be a boolean function which respects an abstraction function $h : B^n \rightarrow N^n$. Let T^f be the levelized BDD for f. In this case, $T^f_h = T^f$.*

Proof: Let p_1 and p_2 be paths of length i. Moreover, assume that $h(p_1) = h(p_2)$. Each node v in the BDD T^f corresponds to a boolean function. We denote the boolean function corresponding to the node v by $b(v)$. Since f respects h, we have $f(p_1) = f(p_2)$. Then the following holds.

$$b(node_{T^f}(p_1)) \quad = \quad b(node_{T^f}(p_2))$$

In BDDs, nodes at the same level which represent the same boolean function are identified. Since T^f is a BDD, $node_{T^f}(p_1) = node_{T^f}(p_2)$. Suppose we execute the algorithm *BDD_DFS* on the levelized BDD T^f. Because of our observation, whenever we call *BDD_DFS* with the arguments $(node_{T^f}(p_1), p_1)$ and $(node_{T^f}(p_2), p_2)$ such that $h(p_1) = h(p_2)$, we are guaranteed that $node_{T^f}(p_1) = node_{T^f}(p_2)$. Hence, the algorithm *BDD_DFS* leaves T^f intact.□

A set of abstraction functions $\{h_1, \cdots, h_p\}$, where $h_i : B^n \rightarrow B^n$ 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 $f : B^n \rightarrow B^n$ be a function and $h : B^n \rightarrow B^n$ be an abstraction function. Given a function $f : B^n \rightarrow B^n$, 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 $h_i : B^n \rightarrow B^n$. 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 $f_i \circ h_j$. Since $f_i \circ h_j$ respects h_j, Lemma 5 implies that $T_h^{i,j} = T^{i,j}$. Let $M(f)$ be a $m \times p$ matrix of aBDDs such that $M(f)_{i,j} = T_h^{i,j} = T^{i,j}$. 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 $f : B^n \rightarrow B^n$ and $g : B^n \rightarrow B^n$ 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 $h_i : B^n \rightarrow B^n$ $(1 \leq i \leq p)$ 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 $M(f) \neq M(g)$. □

As an example, consider the function $mult : B^n \rightarrow B^n$ 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:

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

where x mod p denotes the residue with respect to a positive integer p. 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 $h_i : B^n \rightarrow B^n$ 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 scenario. More generally, this theorem will be true for arithmetic functions with residue abstractions.

4 Equivalence checking using aBDDs

Because of their bounded size, aBDDs can be used to verify the equivalence of large circuits. The general procedure is as follows.

1. Given a circuit, choose a set of appropriate abstraction functions.
2. Select an abstraction function h out of the set.
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 until all abstraction functions in the set have been considered.

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 $f = p \wedge q$. Assume we have already built the aBDD for p and q (with respect to the abstraction function h). Lets call these aBDDs $h(T_p)$ and $h(T_q)$. Next, we build the BDD corresponding to $h(T_p) \wedge h(T_q)$. After that we levelize the BDD and apply our abstraction algorithm to obtain the aBDD for f (technically, applying abstraction is redundant, as shown in lemma 4).

In general, levelizing a BDD can be a very expensive operation. Hence, we have devised a method which does not have to construct the levelized BDD explicitly. If a BDD is reduced instead of levelized, there may be nodes missing along some paths from the root to the terminal nodes. We implicitly visit the missing nodes without creating them and only construct the necessary additional nodes while performing abstraction.

Let us use an example to describe the algorithm. Assume that we have an abstraction function h and circuit in Figure 5. The aBDD associated with line z is $h(T_z)$. At the beginning, let us assume that we have aBDDs at input lines a, b, c, d. By performing the *and* operation, we have BDD $T_e = h(T_a) \wedge h(T_b)$ at line e. Next we perform abstraction on levelized BDD of T_e and obtain the aBDD $h(T_e)$. The same procedure is performed on line f. After we obtain aBDDs on both lines e and f, aBDD for output g can be generated by using the same method.

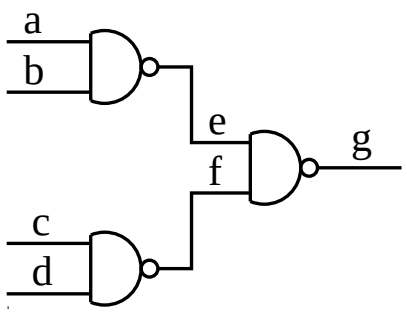

Figure 5: Example:Building aBDD from Circuits

Finally, we describe an optimization which we call *delayed abstraction*. Given a BDD f, building the abstract BDD can be quite expensive. If we have to perform the abstraction after each BDD operation, our algorithm can take a long time. We have modified our algorithm so that we only apply the abstraction operation, when the BDD sizes exceed a certain pre-set limit.

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. Table 1 presents

the comparison of our method to the traditional OBDD method. The results for aBDDs without delayed abstraction are given in the table below. The abstraction function used is

$$a(x_0, \cdots, x_n) = \sum_{i=0}^{n} x_i$$

In Table 1, *Max # Nodes* is the maximum number of BDD nodes that appear 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, c6288 and c7552 are not reported because they exceeded the memory limit.

circuits	Errs	Det.Errs		Max # Nodes		Avg.Time	
		OBDD	aBDD	OBDD	aBDD	OBDD	aBDD
c432	202	202	151	6239	4604	1.45	8.75
c499	288	288	270	96508	9481	22.74	18.35
c880	510	510	299	698891	10507	138.25	66.19
c1355	912	912	843	132984	10296	32.95	47.57
c1908	756	752	629	105424	6386	35.95	26.07
c2670	10	unable	5	–	132593	–	5449.37
c3540	100	100	41	1582309	9927	299.89	107.98
c5315	10	unable	10	–	208795	–	4618.01
c6288	10	unable	6	–	7317	–	86.52
c7552	10	unable	9	–	366462	–	11963.65

Table 1. Comparison between OBDD method and aBDD method

We compare the results with and without delayed abstraction in Table 2. M_1 denotes the aBDD without delayed abstraction. M_2 denotes the aBDD with delayed abstraction and a pre-set node limit of 500, i.e., the abstraction function is only applied to BDDs with more than 500 nodes.

circuits	Errs	Det.Errs		Max # Nodes		Avg.Time	
		M_1	M_2	M_1	M_2	M_1	M_2
c432	10	10	10	4511	4617	8.82	1.49
c880	10	8	10	7589	8167	66.64	27.19
c2670	10	5	6	132593	64363	5449.37	2315.64
c6288	10	6	6	7317	11407	86.52	61.34
c7552	10	9	10	366462	184764	11963.65	2750.45

Table 2. Comparison of Different aBDDs

6 Conclusion

In this paper, we present a new transformation on BDDs, called *abstraction*, which produces BDDs of bounded size. The transformed BDDs are called aBBDs 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 circuits are equivalent, the boolean functions 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. Moreover, we identify an important class of functions for which this technique is complete. This class includes many common arithmetic circuits including integer multiplication.

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] Randal E. Bryant, "Binary Decision Diagrams and Beyond: Enabling Technologies for Formal Verification", *Proc. Intl. Conf. Comput. Aided Design*, pp.236-243, 1995.

[3] Randal E. Bryant, Yirng-An Chen, "Verification of Arithmetic Circuits with Binary Moment Diagrams", *32nd Design Automation Conference*, pp.535-541, 1995.

[4] Edmund M. Clarke, K. L. McMillan, Xudong Zhao, Masahiro Fujita, Jerry C.-Y. Yang, "Spectral Transformation for Large Boolean Functions with Applications to Technology Mapping", *30th Design Automation Conference*, pp.54-60, 1993.

[5] 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.

[6] Edmund M. Clarke, Masahiro Fujita, Xudong Zhao, "Hybrid Decision Diagrams: Overcoming the Limitations of MTBDDs and BMDs", *Proc. Intl. Conf. Comput. Aided Design*, pp.159-163, 1995.

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