A Survey of Abstract BDDs *

Edmund Clarke[1,2] Yuan Lu[2] Helmut Veith[1,3]

[1] Computer Science Dept.
Carnegie Mellon University
Pittsburgh, PA 15213
U.S.A.

[2] Electrical and Computer Eng. Dept.
Carnegie Mellon University
Pittsburgh, PA 15213
U.S.A.

[3] Inst. of Information Systems
Vienna University of Technology,
1040 Vienna
Austria

Abstract

We survey and unify recent work about abstraction within BDD packages. Abstract BDDs (aBDDs) are obtained from ordinary BDDs by merging BDD nodes whose abstract values coincide. We discuss four types of abstract BDDs (called S-type, 0-type, 1-type and $\vee$-type aBDDs) which have found applications in many CAD-related areas such as equivalence checking, variable ordering and model checking. Experimental results demonstrate the usefulness of our method.

1 Introduction

The remarkable success of symbolic verification techniques is closely tied to the introduction of Binary decision diagrams (BDDs) [3]. Since BDDs are often exponentially succinct, they have been successfully used to overcome the state explosion problem for medium size circuits. However, for large systems, the state-explosion problem turns into a *node explosion problem* because the BDDs themselves become too large. Numerous techniques have been developed in order to alleviate this problem [2, 13].

Abstraction techniques are used in hardware verification in order to abstract away from irrelevant behavior, in particular repetitive behavior, and thus to reduce the size of the system. Abstraction *functions* provide a natural way to define abstractions. An abstraction function h is used to map states into a smaller set of abstract states which is intended to be representative for the overall system. For example, many researchers use the modulus function to verify multipliers. Clarke, Grumberg and Long [6] use the Chinese Remainder Theorem to model-check a sequential multiplier. Kimura [9] introduced residue BDDs to verify combinational multipliers. In [12] Ravi *et al.* discuss how to use residue ADDs to verify multipliers efficiently.

Abstract BDDs are generalization of residue BDDs by allowing other abstraction functions besides modulus functions [8]. Other types of abstract BDDs are discussed in [5, 10] as well. In this paper, we summarize these different types of abstract BDDs and show how to apply aBDDs to three areas: equivalence checking, variable ordering for combinational circuits and model checking. The experimental results demonstrate that the great performance improvements can be achieved by using aBDDs. We believe that aBDDs are applicable to many other CAD areas as well.

This paper is organized as follows. Detailed descriptions of the four types of abstract BDDs are provided in Section 2. Section 3 describes different applications of these aBDDs. Conclusion and future research are discussed in Section 4.

2 Preliminaries

We assume that there are n variables $x_1, \cdots, x_n$ over domain $D = B^k$ where $B = \{0, 1\}$. Elements of D are called *vectors* or *states*. In other words, the values assumed by each x_i are the 0-1 vectors from D. For example, for $k = 4$, x_i may have value $\langle 1001 \rangle$. The concatenation of two 0-1 vectors d_1 and d_2 is written as $d_1 \cdot d_2$. For each variable x_i, there is an abstract state variable $\widehat{x_i}$, which takes values in a finite set A called the *abstract domain*. For each x_i there is an *abstraction function* h_i, i.e., a *surjection* $h_i : D \to A$. For the sake of simplicity, we shall often restrict our attention to the case of a single variable x_i and a single abstraction function h_i. When it is clear from the context, we will suppress the index and write the abstraction function simply as $h : D \to A$. The abstraction function h induces an equivalence relation $\equiv_h$ on D as $(d_1 \equiv_h d_2) \leftrightarrow h(d_1) = h(d_2)$. For an element $d \in D$, $[d] = \{d' : d \equiv_h d\}$ denotes the equivalence class of d. The set of equivalence classes of D with respect to the equivalence relation $\equiv_h$ is denoted by $[D]_{\equiv h}$ and defined as $\{[d] : d \in D\}$. For each h, we fix a function $rep_h : [D]_h \to D$ that selects a *unique representative* from each equivalence class $[d]$. Thus, for a 0-1 vector $d \in D$, $rep_h([d])$ is the unique representative of the equivalence class of d.

The abstraction function h induces a new abstraction function $\mathcal{H} : D \to D$ as $\mathcal{H}(d) = rep_h([d])$. Since $\mathcal{H}$ operates on D, we call $\mathcal{H}$ the generated *auto-abstraction function*. From the definition of $\mathcal{H}$ it is easy to see that $\mathcal{H}(rep_h([d])) = rep_h([d])$ and $\mathcal{H}(\mathcal{H}(d)) = \mathcal{H}(d)$. Note that the image of D under the function $\mathcal{H}$ is simply the set of representatives. The set of representatives will be denoted by $Img(\mathcal{H})$. Since h was assumed to be a surjection, it follows that $|Img(\mathcal{H})| = |A|$.

It is well-known that a Boolean function $f : D^n \to B$ ranging over $\{x_1, x_2, \ldots, x_n\}$, can be represented by a *binary decision tree* (BDT). If we merge the isomorphic subgraphs of a BDT, we will obtain a levelized BDD (LBDD). Furthermore, if we eliminate the redundant nodes (i.e., whose edges point to the same node), then a BDD is obtained.

In the following, we will discuss how a single abstraction function is applied to a Boolean function $f : D \to B$. It can

*This research is sponsored by the Semiconductor Research Corporation (SRC) under Contract No. 97-DJ-294, the National Science Foundation (NSF) under Grant No. CCR-9505472, and the Max Kade Foundation. One of the authors is also supported by Austrian Science Fund Project N Z29-INF. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of SRC, NSF, or the United States Government.

be easily generalized to the case with multiple abstractions. Due to the page limit, we omit the details which can be found in [8, 5].

An abstract BDD (aBDD) is obtained by collapsing all paths corresponding to the same equivalence class into a single path. While such a collapse leads to a loss of information, the size of the aBDD may be significantly decreased, and certain problems may become feasible.

The above mentioned collapsing operation can be defined in various ways, and thus, different types of aBDDs are obtained. Since the concepts underlying aBDDs are most easily explained using Binary Decision *Trees*, we will usually outline this special case first, and show later how to deal with the general case.

As described previously, the abstraction function h induces an auto-abstraction function $\mathcal{H} : D \to D$. Given a Boolean function $f : D \to B$ and a BDT T_f, let u, v be two leaves of T_f (note that $\vec{u}, \vec{v} \in D$) such that $\vec{u} \equiv_h \vec{v}$ and $\mathcal{H}(\vec{v}) = \vec{u}$, i.e., $\vec{u}$ and $\vec{v}$ are equivalent with respect to $\equiv_h$, and $\vec{u}$ is the representative of their equivalence class. In this case we say that the node u is the representative of v.

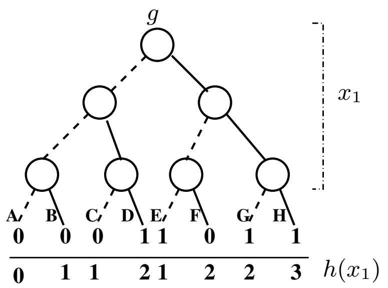


Figure 1: The BDT for g

Example 2.1 Assume that for $D = B^3$ a Boolean function $g : D \to B$ is given by

$$x_1 = \langle 011 \rangle \vee x_1 = \langle 100 \rangle \vee x_1 = \langle 110 \rangle \vee x_1 = \langle 111 \rangle.$$

The BDT for g is depicted in Figure 1. We consider the abstraction function $h(x_1) = count1(x_1)$ which counts the number of 1s in x_1. For example, $count1(\langle 101 \rangle) = 2$. The abstract values for each node are listed in Figure 1. It is easy to see that h induces an equivalence relation $\equiv_h$ on 0-1 vectors of length 3. For example, $\vec{B} \equiv_h \vec{C} \equiv_h \vec{E}$ since they have the same abstract value 1. Assume that the representative of an equivalence class is the lexicographically least vector in that equivalence class. Then $\vec{B}$ is the representative for the equivalence class with respect to the abstract value 1, i.e.,

$$\mathcal{H}(\vec{B}) = \mathcal{H}(\vec{C}) = \mathcal{H}(\vec{E}) = \vec{B}.$$

The representative nodes for the abstract values $0, 1, 2, 3$ are A, B, D, and H respectively. Consider the representative node B which represents B, C, and E. In this case, we need to collapse the paths leading to B, C, and E into a single path leading to B.

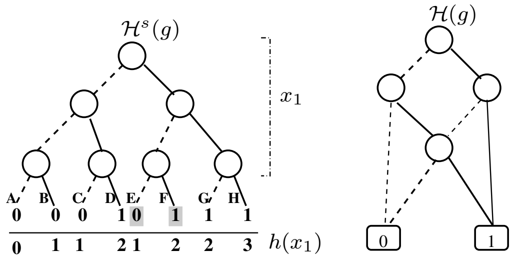
Different operations can be defined for representative nodes and non-representative nodes. This will result in different types of abstract BDDs. In the following subsections, we will define four types of aBDDs: S-type, 0-type, 1-type and $\vee$-type aBDDs.

2.1 S-type Abstract BDDs

Given a Boolean function $f : D \to B$ and a BDT T_f, the S-type aBDT $\mathcal{H}^s(f)$ of T_f rooted at v is defined by

$$\mathcal{H}^s(f)(\vec{v}) = f(\mathcal{H}(\vec{v})).$$

To understand this definition just note that $f(\mathcal{H}(\vec{v}))$ is computed as follows: first, the input vector $\vec{v}$ is transformed into its representative $\mathcal{H}(\vec{v})$, and then, f is applied to the representative. In other words, the output of f for the representative determines the output of f for all the other members of the equivalence class.



(a) BDT for $\mathcal{H}^s(g)$ **(b) aBDD for g**

Figure 2: S-type aBDT and aBDD for g

Example 2.2 For the Boolean function in Example 2.1, the S-type aBDT is shown in Figure 2(a). E is a non-representative node. The value at E is overwritten by the value at B which is a representative of E. A similar situation occurs at nodes C, F, G. The final reduced BDD is in Figure 2(b). Intuitively, the construction maintains some "useful" *minterms* and ignores other "uninteresting" *minterms*.

S-type aBDTs distribute over any logic operations. The following lemma holds.

Lemma 2.1 *Let $f, p, q : D \to B$ be three Boolean functions, $\circ$ be composition and $\odot$ be any logical operation. Then if $f = p \odot q$, then $\mathcal{H}^s(f) = \mathcal{H}^s(p) \odot \mathcal{H}^s(q)$. If $f = p \circ q$, then $\mathcal{H}^s(f) = p \circ \mathcal{H}^s(q)$.*

Assume that u, t are two nodes on level $i + 1$ ($i < k$) in the BDT T_f. As before, $\vec{u}, \vec{t}$ denote the labels concatenated along the paths from the root to u and t respectively. The concatenation $\vec{u} \cdot e$ where $e \in B^{k-i}$ describes a path to a leaf in T_f because $\vec{u} \cdot e \in D$. We define an equivalence relation $\equiv_i$ over B^i (i.e., over 0-1vectors of length i) by

$$\vec{u} \equiv_i \vec{t} \text{ iff } \forall e \in B^{k-i}. h(\vec{u} \cdot e) = h(\vec{t} \cdot e)$$

It is easy to see that $\equiv_i$ is reflexive, symmetric and transitive. Similar to the definition of the auto-abstraction function $\mathcal{H}$, $\mathcal{H}_i(\vec{u})$ is defined to select a unique representative from the equivalence class $[\vec{u}]_{\equiv_i}$. Formally, let $rep_i : [B^i]_{\equiv_i} \to B^i$ be a function which selects a unique representative from $[B^i]_{\equiv_i}$. Then $\mathcal{H}_i(\vec{u}) : B^i \to B^i$ is defined as

$$\mathcal{H}_i(\vec{u}) = rep_i([\vec{u}]_{\equiv_i}).$$

Note that $\equiv_k$ coincides with $\equiv_h$, i.e., we can view the equivalence relations $\equiv_1, \equiv_2, \ldots$ as approximations of $\equiv_h$.

Next, we show how to compute an S-type aBDT from a given BDT. Without loss of generality, assume that each equivalence class is represented by its lexicographically minimal element. Therefore, if $\leq$ denotes lexicographical ordering, and $\mathcal{H}(\vec{w}) = \vec{w}$, then $\vec{w}$ is the lexicographically first vector in $\{\vec{v} : \vec{v} \equiv_h \vec{w}\}$.

Note that LBDDs are obtained from BDTs by merging isomorphic subtrees. The algorithm **AbsSame** in Figure 3 build S-type LBDDs directly from LBDDs. In the algorithm, $Sub(v)$ denotes the subgraph rooted at the node v. $Sub(v) \approx Sub(v')$ means that the respective subgraphs rooted at v and v' are isomorphic. The initial call of the algorithm is $\mathbf{AbsSame}(root_f, \langle\rangle)$.

function AbsSame$(v, path)$
$i = level(v) - 1$;
$\vec{w} = \mathcal{H}_i(path)$;
if $\vec{w} \neq path$
    return w; (*)
else
    if $nonterminal(v)$
        $left(v) = \mathbf{AbsSame}(left(v), path \cdot 0)$;
        $right(v) = \mathbf{AbsSame}(right(v), path \cdot 1)$;
        if there exists v_1 in cache such that $Sub(v) \approx Sub(v_1)$
            return v_1;
        endif;
    endif;
    return v;
endif

Figure 3: Modified **AbsSame** for LBDDs

Given an LBDD f, the BDD $\mathcal{H}^s(f)$ obtained from algorithm **AbsSame** is called an abstract LBDD. If we apply the BDD reduction rules on an abstract LBDD, we will obtain an abstract BDD or an aBDD.

Lemma 2.2 *For a given LBDD f, the size of $\mathcal{H}^s(f)$ is less than or equal to $2 + \prod_{i=1}^{n} |[B^i]_{\equiv_i}|$.*

In practice, when $|A| \ll |D|$, i.e., the range of the abstraction function h is much smaller than the domain of h, the abstraction overhead and the resulting S-type aBDD size are usually very small. As a matter of fact, for many interesting abstraction functions (e.g. modulus, count1, logarithm, linear, partition functions), the abstraction overhead is polynomial in $|A|$.

2.2 0-type And 1-type Abstract BDDs

In this subsection, we introduce two new types of aBDDs (0-type and 1-type aBDDs) which have different properties from S-type BDDs. We will first describe the properties on 0-type aBDDs. The 1-type aBDDs have similar properties.

Recall that in the abstraction procedure for S-type aBDDs, non-representative nodes are replaced by representative nodes. In 0-type aBDDs, however, non-representative nodes are replaced by 0-nodes, i.e., the representative nodes remain unchanged, and all others are uniformly set to output 0. Formally, the 0-type abstract BDD $\mathcal{H}^0(f)$ of T_f rooted at v is defined as

$$\mathcal{H}^0(f)(\vec{v}) = (\mathcal{H}(\vec{v}) \leftrightarrow \vec{v}) \wedge f(\vec{v}) = \begin{cases} f(\vec{v}) & \text{if } \mathcal{H}(\vec{v}) = \vec{v} \\ 0 & \text{otherwise} \end{cases}$$

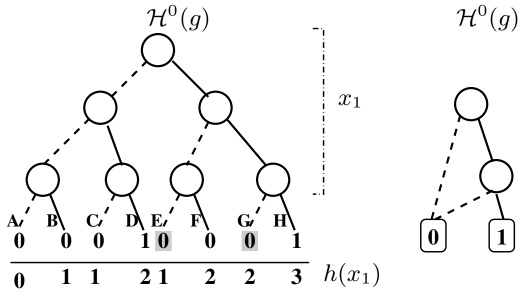


Figure 4: 0-type aBDD for g

Example 2.3 Consider again the Boolean function defined in Example 2.1. The 0-type aBDD for g are shown in Figure 4.

0-type aBDDs do not directly distribute over logic operations. However, the following lemma holds.

Lemma 2.3 *Let $f, p, q : D \to B$ be three Boolean functions, $\circ$ is composition and $\odot$ be any logical operation. Then we have*

$$\begin{array}{lcl} f = p \odot q & \to & \mathcal{H}^0(f) = \mathcal{H}^0(\mathcal{H}^0(p) \odot \mathcal{H}^0(q)) \\ f = p \circ q & \to & \mathcal{H}^0(f) = \mathcal{H}^0(p) \circ \mathcal{H}^0(q) \end{array}$$

The algorithm to obtain 0-type aBDDs can be obtained by modifying the line (*) in **AbsSame** to *return 0*. Lemma 2.2 analogously hold for 0-type abstract LBDDs. Furthermore, the following lemma shows that the 0(1)-type aBDDs can be viewed as lower (respectively upper) approximations of the original function.

Lemma 2.4 *The following tautologies hold.*

$$\mathcal{H}^0(f) \to f \quad \text{and} \quad f \to \mathcal{H}^1(f).$$

2.3 $\vee$-type Abstract BDDs

S-type, 0-type and 1-type aBDDs are similar in the sense that the representative nodes remain unchanged while the non-representative nodes are replaced by new functions.

In this subsection, we define $\vee$-type aBDDs. For $\vee$-type aBDDs, both representative nodes and non-representative nodes are modified. Recall that in a BDT T_f, $\vec{v}$ is the path from the root to node v in a BDT T_f. The $\vee$-type aBDT $\mathcal{H}^\vee(f)$ corresponding to the Boolean function $f : D \to B$ is defined by the following equation:

$$\mathcal{H}^\vee(f)(\vec{v}) = \begin{cases} \bigvee_{\vec{u} \equiv_h \vec{v}} f(\vec{u}) & \text{if } \vec{v} = \mathcal{H}(\vec{v}) \\ 0 & \text{otherwise} \end{cases}$$

Hence, in the obtained aBDT $\mathcal{H}^\vee(f)$, the Boolean function of a representative node is the *disjunction* of all the Boolean functions corresponding to the nodes in the same equivalence class, i.e., the representative node outputs 1 if at least one of its equivalent nodes outputs 1. For non-representative nodes, the Boolean function is defined to be *false* or 0, similar as in 0-type aBDTs.

The next lemma describes how the auto-abstraction function $\mathcal{H}$ interacts with conjunctions and disjunctions.

Lemma 2.5 *Let $f, p, q : D \to B$ be Boolean functions. Then the following tautologies hold:*

$$\begin{array}{lcl} (f = p \vee q) & \rightarrow & (\mathcal{H}^{\vee}(f) = \mathcal{H}^{\vee}(p) \vee \mathcal{H}^{\vee}(q)) \\ (f = p \wedge q) & \rightarrow & (\mathcal{H}^{\vee}(f) \rightarrow \mathcal{H}^{\vee}(p) \wedge \mathcal{H}^{\vee}(q)) \end{array}$$

Lemma 2.6 *Given a Boolean function $f(x)$ and an abstraction function $\mathcal{H} : D \to D$, the following formula holds: $\mathcal{H}^{\vee}(f) = \exists x[\mathcal{H}(x) = y \wedge f(x)]$.*

Lemma 2.6 will be the foundation for the results of section 3.3. In the next section, we will discuss how to apply different types of aBDDs in different situations.

3 Applications of Abstract BDDs

In this section, we will discuss three applications of abstract BDDs: equivalence checking, dynamic reordering and model checking.

3.1 Equivalence Checking Using S/0/1-type Abstract BDDs

For many abstraction functions, aBDDs are usually much smaller than the BDDs of the original function. Moreover, aBDDs maintain partial information of the original Boolean functions. They can be used as a sufficient condition to check *in*-equivalence of large combinational circuits.

Lemma 3.1 *Given two Boolean functions $f, g : D^n \to B$, $\mathcal{H}^s(f) \neq \mathcal{H}^s(g) \to f \neq g$ and $\mathcal{H}^0(f) \neq \mathcal{H}^0(g) \to f \neq g$ and $\mathcal{H}^1(f) \neq \mathcal{H}^1(g) \to f \neq g$.*

When the BDDs for a large multiple-output combinational circuit become extremely large, aBDDs can usually be built instead.

We have implemented 0-type aBDDs into the CMU BDD package and performed a set of experiments on the ISCAS85 benchmark circuits using 0-type aBDDs. Design errors in the circuit were injected one by one by selecting a stuck-at fault on one input of an arbitrary gate. We choose $D = B^{10}$ and the abstraction function $h(x_1) = count1(x_1)$. The abstraction function for the other variables is the identity abstraction function, i.e., $h(x_i) = x_i$ for $i > 1$. In the experiments, the aBDD based approach can detect over 90% of the errors.

Circuit	Errs	Detected Errors		BDD Size	
		OBDD	aBDD	OBDD	0-type
c432	10	10	10	4403	3307
c499	10	10	10	16850	12767
c880	10	10	10	12905	5077
c1355	10	10	10	28582	16661
c1908	10	10	9	13153	7737
c2670	10	10	10	28926	16643
c3540	10	9	5	67586	12790
c5315	10	10	10	17905	13677
c6288	10	–	10	—	8128
c7552	10	10	10	13637	16889

Table 1: Checking inequivalence using 0-type aBDDs

3.2 Improving Variable Ordering Using 0-type Abstract BDDs

Let $f : B^m \to B$ be a Boolean function over the Boolean variables $y_1, y_2, \ldots, y_m$. A *cube* c_i is a monomial over the variables $y_1, \cdots, y_k$ $(k < m)$. *Cube based sampling* [7] partitions the domain of f into smaller cubes $c_1, \cdots, c_{2^{m-k}}$ and uses dynamic variable ordering to select a good ordering for the restriction $f_i = f \wedge c_i$. The ordering for f is obtained by combining the orderings of several randomly chosen f_i. The quality of the resulting ordering may not be very good if f_i does not closely approximate f. Thus, if the subset of cubes is selected randomly, there may be significant variance in the approximations. Consequently, the final ordering for f may not be good.

This problem can be alleviated by using a new sampling technique. Instead of analyzing *one* random cube, we automatically consider multiple cubes at the same time by using 0-type aBDDs. We call this new technique *window based sampling*.

Intuitively, a window is a union of some number of cubes. Assume that we choose t disjoint windows $w_1, \ldots, w_t$. Hence, we can partition f into $f_1, \ldots, f_t$, where $f_i = f \wedge w_i$. In our window based approach we choose the sampling windows using 0-type aBDDs.

According to our definition, we know that $x_1 \in D = B^k$. Setting $n = \lfloor m/k \rfloor + 1$, we can rewrite the Boolean function $f(y_1, \ldots, y_m)$ as $f(x_1, \ldots, x_n)$, where $x_i = \langle y_{(i-1)k+1}, \ldots, y_{i*k} \rangle$ for $i = 1, \ldots, n-1$ and $x_n = \langle y_{(n-1)k+1}, \ldots, y_m, 0 \ldots 0 \rangle$. Let $d = \langle a_1, \ldots, a_k \rangle$ be a k-bit vector where $a_i \in B$ and $d \in D = B^k$. It is easy to see that d induces a cube c_d where

$$c_d = \bigwedge_{i=1}^{k} \begin{cases} y_i & a_i = 1 \\ \neg y_i & a_i = 0 \end{cases}$$

Given an abstraction function h, assume that $h_1 = h$ and $h_i(x_i) = x_i$ for $i > 1$. In other words, abstraction is only applied to the top variable x_1. The other variables are kept unabstracted. If we define a window $w_{\mathcal{H}} = \cup_{\vec{u} = \mathcal{H}(\vec{u})} c_{\vec{u}}$, then the following lemma holds.

Lemma 3.2 *Let f be a Boolean function and $\mathcal{H}$ be an auto-abstraction function. Then $\mathcal{H}^0(f) = f \wedge w_{\mathcal{H}}$.*

According to Lemma 3.2, using 0-type aBDDs provides a natural way to implement the window based sampling method. First, we select a set of *control variables*. These variables are heuristically determined by traversing the circuit in a depth-first order where nodes are selected so that the distance from a node to the primary inputs is minimized. Next, we choose an abstraction function for a set of control variables and build an abstract BDD for the function f with dynamic reordering. Since this abstract BDD partially captures the functionality of f, a good ordering for the abstract BDD is likely to be a good ordering for f as well. Different abstraction functions usually produce different orders. From our experiments, we have found that the *count1* function $\sum_{i=1}^{k} y_i$ and the *logarithmic* abstraction function $\lfloor \log_2 \sum_{i=1}^{k} (2^i y_i) \rfloor$ are good choices. Note that these abstraction functions are parameterized by the number of variables. In each case, the number of cubes is relatively small. For example, a count1 abstraction function on k variables determines $k+1$ cubes.

A function sampled using windows effectively contains a restriction of the original function on each of the cubes. Thus, when we reorder our sampled function, we are implicitly trying to produce an order which is simultaneously "good" for each of these restrictions. Intuitively, this is important because a variable order produced from restriction by any single cube may not be good for the whole function. Considering multiple cubes at the same time and "averaging" their effect is more likely to produce better results. For many circuits we find that the variable order produced by using windows is far better than the order produced by cubes.

Our experiments are performed on a 360MHz Sun UltraSparc-60 with 512Mb RAM using the CUDD-2.2.0 package. In our table, BDD size is measured by the number of BDD nodes. Runtime entries refer to the time taken for the sampling phases, as well as the time taken to construct the final BDD from the order computed by sampling. All CUDD entries refer to CUDD-2.2.0 using *sift*, The "Using aBDD" columns refer to the sampling technique which uses 0-type abstract BDDs. We conducted two sets of experiments. The first experiment shows how the technique behaves on single output functions, while the second experiment deals with multiple output functions.

Experiment 1 (Figure 5) demonstrates the utility of window based sampling compared with cube-based sampling scheme. The benchmarks clearly show that our method does not have the large deviation problem.

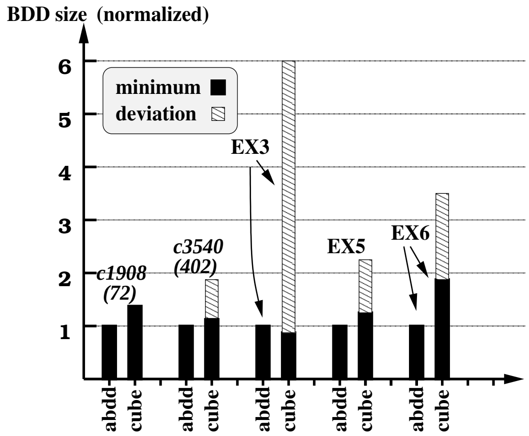


Figure 5: Dynamic ordering with aBDD vs. cube based method

Experiment 2 (Table 2) is another set of experiments that verify the efficiency of window based methods on multiple output functions. It is known that sifting works very well for ISCAS85 circuits [11] and for many circuits, there may not be scope for significant improvement. However, our approach still outperforms CUDD for some of the circuits (c1908 and c7552). For large industrial circuits, our approach is definitely much better than CUDD (*sift*) with respect to both time and space.

3.3 Model Checking Using $\vee$-type Abstract BDDs

Given a Kripke structure $M = (S, S_0, R)$, where S is the set of states, $S_0 \subseteq S$ is the set of initial states, and $R \subseteq S \times S$ is the transition relation, the abstract structure $M_h = (S_h, S_{0,h}, R_h)$ is defined as follows:

$$S_{0,h} = \exists x_1 \cdots [h(x_1) = \hat{x}_1 \wedge S_0(x_1, \cdots, x_n)]$$
$$R_h = \exists x_1 \cdots \exists x'_1 \cdots [h(x_1) = \hat{x}_1 \wedge \cdots h(x'_1) = \hat{x'}_1 \cdots \wedge R]$$

	CUDD Sift		Using aBDDs	
Ckts	BDD Size	CPU Time	BDD Size	CPU Time
c432	1246	0:02	1224	0:03
c499	25897	0:29	26798	1:03
c880	4821	0:06	4463	0:06
c1355	25897	0:31	26579	0:56
c1908	**9102**	**0:07**	**5946**	**0:08**
c2670	2412	0:15	3070	0:31
c3540	23857	0:27	24122	1:02
c5315	2108	0:06	2712	0:07
c7552	**18363**	**2:26**	**7206**	**0:59**
M1	**2595K**	**1:54:45**	**1866K**	**1:26:41**
M2	**4283K**	**8:36:00**	**4120K**	**2:50:15**
M3	**963K**	**1:17:15**	**487K**	**28:49**
M4	**fail**	**fail**	**2195K**	**1:13:26**
M5	**5976**	**0:48**	**1568**	**2:23**
M6	**89639**	**4:24**	**13625**	**2:36**

Table 2: Sampling using 0-type aBDD for ISCAS85 circuits

where $S = D^n$ and x_i is a symbolic variable over D. On the other hand, we define another abstract structure $M_H = (S_H, S_{0,H}, R_H)$, which we construct using $\vee$-type aBDDs:

- The *state set* S_H is the image of S under the auto-abstraction function $\mathcal{H}$.
- The *initial set of states* $S_{0,H}$ is the image of S_0 under the function $\mathcal{H}$. Notice that if S_0 is represented as a boolean function, then $S_{0,H}$ corresponds to the $\vee$-type aBDD $\mathcal{H}(S_0)$ (see Lemma 2.6).
- The *transition relation* R_H is the image of R under the function $\mathcal{H}$. Notice that if R is represented as a boolean function, then R_H corresponds to the $\vee$-type aBDD $\mathcal{H}(R)$ (see Lemma 2.6).

Lemma 3.3 $M_h \cong M_H$

Lemma 3.3 states that M_h and M_H are *isomorphic structures*, i.e., there is a bijection $u : S_h \rightarrow S_H$ such that $u(S_0) = S_{0,H}$ and $u(R_h) = R_H$. Therefore, $\vee$-type aBDDs can be used to build abstract Kripke structures. In the following, we will discuss how to use the obtained abstract Kripke structure for generating good initial variable orderings for model checking.

Many static ordering approaches for model checking have been proposed [1]. Because the best ordering may change dynamically during the fixpoint computation, these approaches are not powerful enough for many applications. In reality, people generate the initial orders manually or statically and run model checker iteratively to produce a *golden* variable order. This approach is not systematic and may be inefficient for large designs.

Since the abstract Kripke structure describes the basic behavior of the original structure, a good variable order for the abstract structure is likely to be a good ordering for the orginal one. Based on this observation, we propose a new variable ordering scheme as follows:

1. Given a set of abstraction functions, the system automatically builds the abstract Kripke structure using $\vee$-type aBDDs.

2. Next, we run the model checker on the CTL property with the abstract structure with dynamic reordering on. Counterexample generation is disabled in this phase.

3. Finally, we restart the model checker on the original structure using the ordering obtained from the previous step as the initial variable ordering.

As an example, we verified the PCI bus protocol. The PCI local bus protocol includes three types of devices: masters, targets, and bridges. Masters can start transactions, targets respond to transactions, and bridges connect buses. Masters and targets are controlled by finite-state machines. We considered a simple model which consists of one master, one target, and one bus arbiter. The model includes different timers to meet the timing specification. The master and target both include a *lock* machine to support exclusive read/write. The master also has a data counter to support *burst* transactions (multiple data phases). We have observed that the BDD sizes constructed during model checking can be reduced significantly by using the procedure described above. In our verification phase, we have applied abstractions to some of the timers, the lock machine and the data counter in the master. Address and data in both the master and the target are also abstracted. Various properties dealing with handshaking, read/write transactions, and timing are checked in this model. The experimental results are listed in Table 3. The initial ordering for both "SMV" and "SMV+aBDD" columns are provided manually. Obviously, aBDD based approaches are superior to the traditional approach. Note that our approach is totally automatic.

Property	TIME (sec)		# Nodes	
	SMV	SMV+ aBDD	SMV	SMV+ aBDD
P_1	542	289	11984K	3327K
P_2	242	204	1778K	718K
P_3	5882	207	36077K	862K
P_4	15	77	50K	44K
P_5	424	269	4458K	3700K
P_6	179	118	2472K	520K
P_7	8970	3956	28924K	13964K
P_8	84	117	645K	504K
P_9	9946	793	37288K	5084K
P_{10}	14	75	59K	39K
P_{11}	5580	2713	20680K	7850K
P_{12}	293	376	4632K	3506K
P_{13}	2043	1209	19703K	6002K
P_{14}	2932	1862	38210K	17386K
P_{15}	2831	118	12740K	520K
P_{16}	fail	3955	fail	13964K
P_{17}	63	117	649K	504K

Table 3: Generating Initial Ordering for Model Checking using $\vee$-type aBDDs

4 Conclusion And Future Research

In this paper, we introduce a general framework for applying abstraction to BDDs. We described four types of abstract BDDs and their applications. Other types of abstract BDDs can be easily defined following a similar procedure. Experimental results show that these types of aBDDs are very useful for many applications.

We are currently pursuing several directions for future research. First, we intend to try our techniques on larger designs. Second, we are working on a method for generating and refining abstraction functions automatically. Recent results in this direction have been reported in [4].

References

[1] A. Aziz, S. Tasiran, and R. K. Brayton. BDD variable ordering for interacting finite state machines. In *Design Automation Conference*, 1994.

[2] K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient implementation of a BDD package. In *Design Automation Conference*, pages 40–45, 1990.

[3] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transaction on Computers*, pages 35(8):677–691, 1986.

[4] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. Technical Report CMU-CS-00-103, Computer Science, Carnegie Mellon University, 2000.

[5] E. Clarke, S. Jha, Y. Lu, and D. Wang. Abstract BDDs: a technique for using abstraction in model checking. In *Correct Hardware Design and Verification Methods*, volume 1703 of *LNCS*, pages 172–186, 1999.

[6] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and System (TOPLAS)*, 16(5):1512–1542, September 1994.

[7] J. Jain, W. Adams, and M. Fujita. Sampling schemes for computing OBDD variable orderings. In *International Conference of Computer-Aided Design*, 1998.

[8] S. Jha, Y. Lu, M. Minea, and E. Clarke. Equivalence checking using abstract BDDs. In *International Conference of Computer Design*, 1997.

[9] S. Kimura. Residue BDD and its application to the verification of arithmetic circuits. In *Design Automation Conference*, 1995.

[10] Y. Lu, J. Jain, E. Clarke, and M. Fujita. Improving BDD variable ordering using abstract BDDs and sampling. In *Proceedings of Intl. Workshop of High Level Deisgn, Verification and Testing*, 1999.

[11] S. Panda and F. Somenzi. Who are the variables in your neighborhood. In *International Conference of Computer-Aided Design*, 1995.

[12] K. Ravi, A. Pardo, G. D. Hachtel, and F. Somenzi. Modular verification of multipliers. In *Formal Methods in Computer-Aided Design*, pages pp.49 – pp.63, November 1996.

[13] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *International Conference of Computer-Aided Design*, 1993.