Efficient Variable Ordering Using aBDD Based Sampling

Yuan Lu†, Jawahar Jain‡, Edmund Clarke†, Masahiro Fujita‡

Dept of Elect. and Comput. Eng.†
Carnegie Mellon University
yuanlu,emc@cs.cmu.edu

Advanced CAD Research‡
Fujitsu Laboratories of America
jawahar,fujita@fla.fujitsu.com

Abstract

Variable ordering for BDDs has been extensively investigated. Recently, sampling based ordering techniques have been proposed to overcome problems with structure based static ordering methods and sifting based dynamic reordering techniques. However, existing sampling techniques can lead to an unacceptably large deviation in the size of the final BDD. In this paper, we propose a new sampling technique based on abstract BDDs (aBDDs) that does not suffer from this problem. This new technique, easy to implement and automate, consistently creates high quality variable orderings for both combinational as well as sequential functions. Experimental results show that for many applications our approach is significantly superior to existing techniques.

1 Introduction

OBDDs (Ordered Binary Decision Diagrams) [2] often determine the performance of tools used in synthesis, verification, validation, etc. Variable ordering is the central problem in using BDDs effectively. Numerous heuristics have been proposed to address this problem. *Topology based* or *static* variable ordering techniques (for example, using depth-first or breadth-first search) have been extensively investigated for more than a decade [5, 9]. However, these techniques often perform poorly due to their reliance on purely structural information. *Sifting-based* dynamic ordering techniques are more popular [6, 11, 12], but they are extremely expensive in both time and space. Moreover, during the reordering of the BDDs, these techniques can frequently get stuck in a local minimum and thus fail to reduce the size of the resulting graph to an acceptable degree.

A sampling based solution has been proposed by Jain, et al. [7] to overcome these problems. In their approach, a portion of Boolean space for the output function is analyzed using reordering techniques. This order is then used for analyzing the complete Boolean space of the given function. By appropriately using the (limited) global information about the given function, the local minimum problem of current sifting-based ordering techniques was observed to be significantly reduced. We found this sampling approach to be conceptually very useful. The framework laid out in [7] closely guides various steps of our procedure too. However, even though they discuss the possibility of using sophisticated sampling functions, their implementation uses only *randomly generated cubes* for sampling. Sampling by randomly generated cubes has several practical problems. First, it appears to provide less efficient variable orders. Secondly, variable orders generated can vary dramatically between different runs. This causes an extremely large variance in the quality of the results. This also makes cube based sampling difficult to automate effectively. Also, sampling based on randomly generated cubes appears less effective in producing a common order for multiple output functions.

In this paper we propose a new sampling methodology, which alleviates these problems. Our algorithm uses a deterministic approach based on *abstract BDDs* (aBDDs) [8] and is significantly more efficient in time and space than existing techniques. We describe our technique and explain its advantages in detail in Section 2. We provide detailed experimental results for both combinational and sequential circuits in Section 4, and our conclusions in Section 5.

2 Window-based Sampling Using aBDDs

Let f be a boolean function over the variables $x_1, \cdots, x_n$. A *cube* c_i is just a monomial over the variables $x_1, \cdots, x_m$. *Cube based sampling* [7] partitions the domain of f into smaller cubes $c_1, \cdots, c_{2^{n-m}}$ and uses dynamic variable ordering to select a good ordering for 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 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.

We overcome this problem by using a new sampling technique. Instead of analyzing *one* random cube, we automatically consider multiple cubes at the same time by using abstract BDDs. We call our 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 abstract BDDs.

In order to derive a window, we will divide the set of 2^n input vectors into a number of equivalence classes. We will then construct the representation for the given Boolean function by taking only 1 representative vector from each equivalence class. Our window will be intuitively defined by the set of such representative vectors. More formally, assume that we are given a *surjection* $h : D \rightarrow A$. The function h determines an *abstraction function* for m boolean variables, where $D = \{0, 1\}^m$. The elements $d, d_1, d_2 \in D$ are 0-1 vectors with length m. An equivalence relation $\equiv_h$ is induced by h on D as follows:

$$(d_1 \equiv_h d_2) \leftrightarrow h(d_1) = h(d_2).$$

The set of all possible equivalence classes of D under the equivalence relation $\equiv_h$ is denoted by $[D]_h$ and defined as: $\{[d] | d \in D\}$ where $[d]$ denotes the equivalence class for d. Assume that we have a function $rep : [D]_h \rightarrow D$ that selects a unique representative from each equivalence class $[d]$. In other words, for a 0-1 vector $d \in D$, $rep([d])$ is the unique representative in the equivalence class of d. Moreover, the abstraction function h generates an abstraction function $\mathcal{H} : D \rightarrow D$ as follows: $\mathcal{H}(d) = rep([d])$. Intuitively, $\mathcal{H}$ maps any 0-1 vector to its representative 0-1 vector from the same equivalence class. We call $\mathcal{H}$ the *generated abstraction function*. ¿From the definition of $\mathcal{H}$ it is easy to see that $\mathcal{H}(rep([d])) = rep([d])$.

Intuitively, the aBDDs reduce the sizes of BDDs by using the abstraction function to modify (remove) various paths which do not confirm to the abstraction function. The concepts underlying abstract BDDs are most easily explained using binary decision trees (BDTs) but apply to BDDs as well. For a detailed account, please refer to [8]. Given a BDT T_f for the boolean function f, let $\vec{v}$ denote the path from root to the node v at level $m + 1$. It is easy to see that the path is a 0-1 vector in the domain $D = \{0, 1\}^m$, i.e. $\vec{v} \in D$. As we described before, an abstraction function $h : D \rightarrow A$ induces a generated abstraction function $\mathcal{H} : D \rightarrow D$. Assume that $\vec{u} = rep([\vec{v}])$, i.e. $\mathcal{H}(\vec{v}) = \vec{u}$, we call u the *representative* of node v. Intuitively, in the abstraction procedure, if $\vec{u} = \mathcal{H}(\vec{u})$, the BDT rooted at u is kept; otherwise, the BDT is replaced by "0". More formally, the abstract BDT $\mathcal{H}(f)$ of T_f rooted at v is defined as

$$\mathcal{H}(f)(\vec{v}) \;=\; \begin{cases} f(\vec{v}) & \vec{v} = \mathcal{H}(\vec{v}) \\ 0 & \text{otherwise.} \end{cases}$$

For example, given a boolean function $f = (a \wedge \neg c) \vee (b \wedge c)$, and an abstraction function $h = a + b$, the abstraction procedure is illustrated in Figure 1. First, the BDT for f (Figure 1a) is shown in Figure 1b. We have $\vec{P} \equiv_h \vec{Q}$ since $h(\vec{P}) = h(\vec{Q}) = 1$. Assume that P is chosen as a representative, Then the directed graph

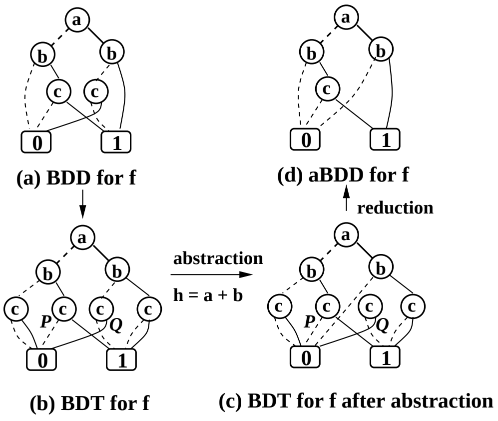

Figure 1: Build abstract BDDs (example)

after abstraction is shown in Figure 1c. Finally, the abstract BDD of f is obtained by applying BDD reduction rules. Note that this new definition of aBDD is different from the one in [8].

The following lemma guarantees that the aBDD of a function f can be obtained by applying abstraction before building f. This avoids building the BDD for the original function f which might be infeasible.

Lemma 2.1 *Let $f, p, q : \{0, 1\}^n \rightarrow \{0, 1\}$ be boolean functions, and let $\mathcal{H} : D \rightarrow D$ be the generated abstraction function corresponding to the abstraction function $h : D \rightarrow A$. The following equations hold:*

$$\begin{array}{lcl} (f = p \circ q) & \rightarrow & (\mathcal{H}(f) = \mathcal{H}(p) \circ \mathcal{H}(q)) \\ (f = \neg p) & \rightarrow & (\mathcal{H}(f) = \mathcal{H}(\neg \mathcal{H}(p)) \end{array}$$

where $\circ$ is either a conjunction or a disjunction.

Assume that the boolean variables in the BDD of f associated with $\vec{v}$ are $x_1, \cdots, x_m$. Let $\vec{v} = \langle a_1, \cdots, a_m \rangle$, $a_i \in \{0, 1\}$. It is easy to see that $\vec{v}$ induces a cube c_v where

$$c_v = \bigwedge_{i=1}^{m} \begin{cases} x_i & a_i = 1 \\ \neg x_i & a_i = 0 \end{cases}$$

Let us define a window $w_{\mathcal{H}} = \bigvee_{\vec{u} = \mathcal{H}(\vec{u})} c_u$. Intuitively, $w_{\mathcal{H}}$ represents a set of cubes. Then we have

Lemma 2.2 *Let f be a boolean function and $\mathcal{H}$ be the generated abstraction function, then $\mathcal{H}(f) = f \wedge w_{\mathcal{H}}$.*

The aBDDs provide us a window in which sampling can be performed. Hence, using abstract BDDs one can naturally 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 on. 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 symmetric abstraction function $\sum_{i=1}^{m} x_i$ and the logarithmic abstraction function $\lfloor \log_2 \sum_{i=1}^{m} (2^i x_i) \rfloor$ 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 symmetric abstraction function on m variables determines $m+1$ cubes.

3 Algorithmic Details

Our method has same 4 conceptual steps as in [7]: the estimation phase, the candidate-order selection phase, the testing phase (circuit filter phase), and the evolution phase. These 4 phases produce an initial ordering for building the final BDD and are described below. The main difference between our approach and [7] is that we select windows using abstract BDDs instead of randomly selected cubes.

Step 1. In the *estimation phase*, we try k ($k \approx 2, 3$) different abstraction functions and determine the number of variables $m_i\,(i \leq k)$ for each function. Starting from the top variable, we choose the set of abstracted variables incrementally. For each cube in the window given by the abstraction function, we partially simulate the circuit. We choose $m_i\,(i \leq k)$ to be the size of the abstraction function if simulating one of the cubes greatly decreases the number of gates left in the circuit.

Step 2. In the *candidate-order selection phase*, we apply k different abstraction functions to the top $m_i\,(i < k)$ variables selected in the previous phase. Then, for each abstraction function, we build the abstract BDDs for the original boolean function with dynamic reordering on. This produces k different variable orderings. In our experiments, we choose k to be 2 or 3 and use the subsequent phases to reject and refine these orderings.

Step 3. The purpose of the *circuit filter phase* is to filter out the bad orderings. We estimate the quality of a given variable ordering by building the BDD with this ordering until a certain target gate inside the circuit (with dynamic reordering disabled). An obvious question is how we choose the target gate. Considering the circuit to be levelized, we first define some threshold level t_l. Now, we pick that gate between the primary inputs and level t_l whose cone covers the maximum number of primary inputs. The intuition for this step is that we want to consider as many variables as possible to compare the orderings for all of the variables obtained from Step 2.

Step 4. After filtering out the bad orderings, we use the *evolution filter* to decide which is the best ordering from the ones that remain. Using another window defined by a new abstraction function, we build abstract BDDs for the remaining orderings obtained from Step 3. We choose the ordering which has the minimum number of BDD nodes as our final order.

3.1 BDD Ordering in Model Checking

In model checking, the problem of generating a good initial variable ordering is even more serious than the case with combinational circuits. Static ordering approaches 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.

In [4], a methodology to verify ACTL properties using an abstract Kripke structure. Recently, a modified version of abstract BDDs has been used to build a more refined abstract Kripke structure [3]. Note that this modified aBDD is different from the one defined in Section **??**. 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 original structure. 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 abstract BDDs.

2. Next, we verify each ACTL property on the abstract structure with dynamic reordering. If the property is not *true,* we output the final variable ordering for next step.

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

Compared with the methodology for combinational circuits, this approach does not have the evolution phase. We are currently trying to devise an evolution phase suitable for model checking.

As an example, we verified part of the PCI bus protocol. 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.

	SPACE (# of BDD Nodes)					TIME (in seconds)				
Ckts	DFS MIN	Static (aBDD)	CUDD Sift	CUDD SiftConv	Using aBDDs	DFS MIN	Static (aBDD)	CUDD Sift	CUDD SiftConv	Using aBDDs
c432	5624	3956	379	377	367	1.6	3.1	1.3	2.8	2.9
c499	3466	3429	3457	3650	3117	0.1	5.1	3.5	7.2	5.3
c1355	3652	3109	2557	3337	3529	0.1	5.0	3.2	11.0	6.9
c1908	2187	1428	**901**	**758**	**763**	0.2	2.6	**2.0**	**4.5**	**2.6**
c3540	**55730**	**6976**	**8045**	**5486**	**5510**	**9.1**	**30**	**46.0**	**54.0**	**31.0**
c6288	19417	22360	16774	16693	16746	5.1	132	40.0	110.0	56.0
c6288	48483	42781	40024	39942	40024	17.0	127	88.0	251.0	103.0
EX1	**fail**	**942**	**1467**	**644**	**748**	**fail**	**88**	**41**	**89**	**33**
EX2	881339	596415	**13390**	**14771**	**9431**	9.5	24	**22**	**98**	**33**
EX3	966210	738906	**633780**	**655556**	**63404**	8.8	91	**1320**	**6780**	**230**
EX4	fail	fail	**163854**	**fail**	**130589**	fail	fail	**3535**	**fail**	**2667**
EX5	fail	fail	**190674**	**190674**	**63916**	fail	fail	**2616**	**2586**	**480**
EX6	**fail**	**20994**	**20343**	**15905**	**13457**	**fail**	**134**	**146**	**334**	**120**
EX7	fail	fail	**118378**	**67384**	**40698**	fail	fail	**522**	**517**	**191**
EX8	fail	fail	**289619**	**387116**	**186754**	fail	fail	**786**	**4781**	**1365**

Table 1: Deterministic sampling using aBDD (static and dynamic): notice that ISCAS85 circuits like c880, c2670, c5315, c7552, have no hard outputs and hence they are ommitted from the single output table.

3.2 Advantages of our technique

In a cube based sampling technique, since only one cube is considered at a given time, a sample may map to a trivial function. A window based sampling method considers a large number of cubes at one time; it is highly unlikely that each of these cubes will reduce to a trivial function. Thus, even if random cubes were generated, a window based sampling is far more stable. As a corollary, since cube based sampling is very sensitive to the set of cubes generated, this type of technique is hard to automate.

Note a window contains many cubes. Thus, a function sampled using windows effectively contains a restriction of the original function on each of the cube. 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 result. For many circuits we find that the variable order produced by using windows is far better than the order produced by cubes. When a single order is needed for all outputs of a multiple-output circuit, window based method can also be used to generate good initial variable orderings for such circuits.

4 Experimental Results

Our experiments are performed on a 360MHz Sun UltraSparc-60 with 512Mb RAM using the CUDD-2.2.0 package for combinational circuits; and on a 200MHz Pentium-Pro with 1.0Gb RAM using SMV [10] for model checking. In our tables, BDD size

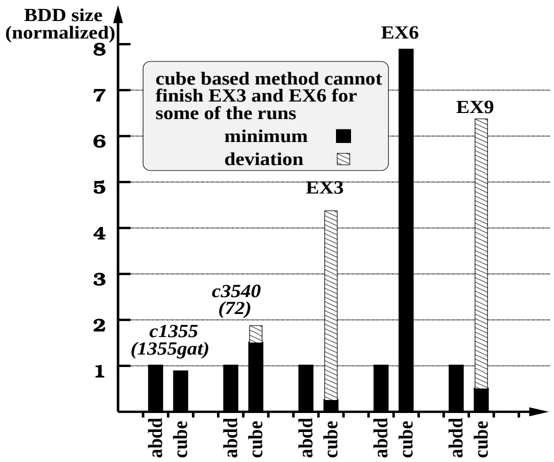

Figure 2: Static ordering using aBDD based sampling vs. using cube based sampling method [7]

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. The "DFS-MIN" entries refer to the DFS based static variable ordering method described in Section 2. Similarly, all CUDD entries refer to CUDD-2.2.0 using *sift*, except for "CUDD SiftConv" which was obtained by replacing *sift* with *sift-convergence* throughout the experiment. The "SMV" column refers to SMV-2.4.4 using partitioned transition relations with 2000 nodes as the partition size. The "Using aBDD" column refers to the sampling technique which uses abstract BDDs for building the abstract structure. We conducted four sets of experiments. Experiments 1-3 use combinational circuits while Experiment 4 deals with model checking. Experiments 1-2 show how the technique behaves on single output functions, while Experiment 3 deals with multiple output functions.

Note, our abstract BDD method gives deterministic results (unlike [7]). For this purpose, in Experiments 1-3 we use two abstraction functions: the *symmetric* function and the *logarithmic* function (See Section 2).

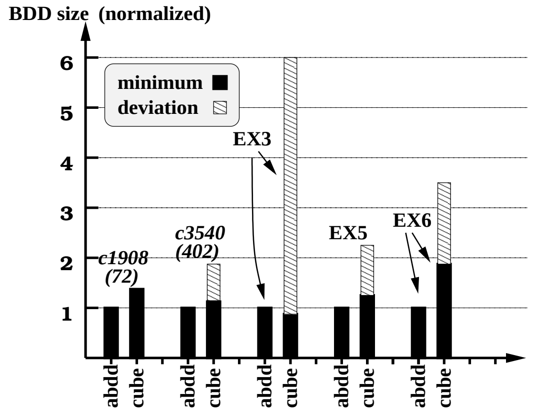

Figure 3: Dynamic ordering using aBDD based sampling vs. using cube based sampling method [7]

Experiment 1 (Table 1 and Figure 2): First, we use the order computed by sampling to build the BDD statically. Except for slightly inferior orderings on c499 and c1355 (both circuits are functionally equivalent) we find that our methods always produce better variable orderings than those produced by DFS search based static techniques (Table 1). For many industrial examples we find that DFS-MIN cannot even process the circuits. Interestingly, for c3540 and EX1, we find that our static order using abstract BDD based windows is better than even the dynamic ordering obtained using the CUDD-2.2.0 package, and for EX6, comparable. Thus, we believe that our window based sampling method is superior to other static ordering methods in terms of efficiency as well as stability.

Figure 2 gives some representative data for comparing the performance of static ordering methods that use an initial ordering provided by cube based sampling vs. window based sampling using aBDDs. It is easy to see that cube based method suffers from very large variance. However, since window based sampling is deterministic, there is no variance at all. Interestingly, for EX3 and EX6, aBDD based methods can create a small BDD for the output function, but cube based sampling fails for some of the runs!

	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: aBDD Sampling for multi-output circuits. Note, multiplier c6288 is ommitted since it is provably intractable for OBDDs.

Experiment 2 (Table 1 and Figure 3) show the utility of window based sampling in a dynamic variable ordering scheme. That is, we show how dynamic reordering techniques can be significantly improved if they are supplied with an initial variable ordering generated using a window based sampling technique. In Table 1, we find that we can produce far smaller graphs than the traditional dynamic reordering methods (*sift, sift-convergence*). Also, for most of the large circuits we take less time. Sometimes, the difference is dramatic; in EX3 we take almost an order of magnitude less space and 6 times less runtime. Compared with cube based sampling approaches, our method is also superior (Figure 3) since our method does not have the large deviation problem.

Experiment 3 (Table 2): We performed another set of experiments to show the efficacy 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*) in both time and space.

Experiment 4 (Table 3): During verification of the PCI bus protocol, 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. Var-

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: Sampling for Model Checking

ious properties dealing with handshaking, read/write transactions, and timing are checked in this model. The initial ordering for both "SMV" and "Using aBDD" columns are provided manually. Obviously, aBDD based approaches are superior to the traditional approach. Note that our approach is totally automatic.

5 Conclusion and Future Work

We have described a highly effective sampling based ordering technique. Our technique is very easy to automate, and provides efficient solutions for both static variable ordering as well as dynamic variable ordering problem. Our results show significant improvement over CUDD package as well as over the previously reported sampling techniques which have the disadvantage of large variations among multiple runs in the quality of results produced. We show that similar approaches using abstraction work very well for model checking too.

We are currently pursuing several directions for future research. As we discussed before, we found that in practice the symmetry and logarithm functions are good abstraction functions. We are currently exploring this aspect of sampling based variable ordering problem; a proper understanding of the same can be extremely useful in generating better abstraction functions automatically. BDD packages with Sampling based variable ordering have been investigated previously by [13]. We are currently investigating how to incorporate our techniques inside BDD packages instead of exploring the circuit structure.

References

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

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

[3] 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.

[4] 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.

[5] M. Fujita et al. Evaluation and improvements of Boolean comparison method based on binary decision diagrams. In *International Conference of Computer-Aided Design*, 1988.

[6] M. Fujita et al. On variable ordering of Binary Decision Diagrams for the application of multi-level logic synthesis. In *European Design Automation Conference*, 1991.

[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. Malik et al. Logic verification using binary decision diagrams in a logic synthesis environment. In *International Conference of Computer-Aided Design*, 1988.

[10] K. L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.

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

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

[13] A. Slobodova and C. Meinel. Sample method for minimization of OBDDs. In *IWLS'98*, pages 311–316, 1998.