Improving BDD Variable Ordering Using Abstract BDDs and Sampling

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

Carnegie Mellon University[†],
5000 Forbes Ave,
Pittsburgh, PA 15213

Fujitsu Laboratories of America[‡]
595 Lawrence Expressway,
Sunnyvale, CA 94086

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 sampling technique based on abstract BDDs (aBDDs) that does not suffer from this problem. This new technique has been implemented and 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, but they are extremely expensive in both time and space. Moreover, they frequently get stuck in a local minimum and fail to reduce the size of the OBDD to an acceptable degree.

A sampling based solution has been proposed by Jain, et. al. [6] to overcome these problems. Their technique is conceptually useful, but suffers from several serious problems. Their technique is difficult to automate effectively. Also, their algorithm uses a randomized approach that may give an extremely large variance in the quality of the results. It is also difficult to handle multiple output circuits simultaneously.

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 3, and our conclusions in Section 4.

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* [6] 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. [1]

[1] Windows can be selected by other analysis methods as well,

2.1 Using aBDDs for Sampling: Details

Before describing the abstract BDDs, we provide some definitions. Assume that we are given a *surjection* $h : D \rightarrow A$. The function h determines an *abstraction function* for m boolean variables, where $D = \{0,1\}^m$. Assume that $d, d_1, d_2 \in D$ 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: $\{[d]|d \in D\}$. 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: $\mathcal{H}(d) = rep([d])$. We call $\mathcal{H}$ the *generated abstraction function*. From the definition of $\mathcal{H}$ it is easy to see that $\mathcal{H}(rep([d])) = rep([d])$.

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 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 we can build the abstract BDD for a function without building the original BDD first.

for example by using decomposition points [7]. Such techniques have been implemented and tested with a high degree of success. For brevity, they will not be discussed in this paper.

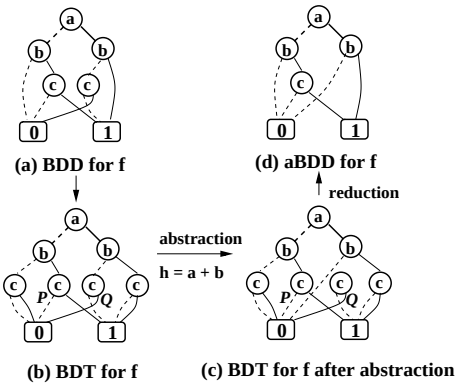

Figure 1: Build abstract BDD (example)

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: $(f = p \circ q) \rightarrow (\mathcal{H}(f) = \mathcal{H}(p) \circ \mathcal{H}(q))$ and $(f = \neg p) \rightarrow (\mathcal{H}(f) = \mathcal{H}(\neg\mathcal{H}(p))$, 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 $\vec{v} =< a_1, \cdots, a_m >$, $a_i \in \{0,1\}$. 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 $w_{\mathcal{H}} = \cup_{\vec{u}=\mathcal{H}(\vec{u})} c_u$. Then we have

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

According to Lemma 2.2, using abstract BDDs 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 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.

Our method has 4 steps: 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.

Step 1. In the *estimation phase*, we try k different abstraction functions and determine the number of variables in each. Starting from the top variable, we choose the set of abstracted variables $m_i (i \leq k)$ incrementally. For each cube in the window given by the abstraction function, we partially simulate the circuit. We choose m_i 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, we build the abstract BDDs for the original boolean function with dynamic reordering on. Each produces a new variable ordering. 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 up to a certain target gate inside the circuit (with dynamic reordering disabled). An obvious question is how we choose the target gate. Using some threshold level, we pick the gate between the primary inputs and this threshold level 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. This idea is also discussed in [6]. The difference is that we select windows using abstract BDDs instead of randomly selected cubes.

2.2 Variable 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. Many 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 [3], the authors propose 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 [4]. Note that this modified aBDD is different from the one defined in Section 2.1. 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 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 on. If the property is not *true,* we reserve 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 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)

2.3 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.

The advantages of a window based method are particularly impressive when a single order is needed for all outputs of a multiple-output circuit. In fact, if we rely on cubes alone, then since the control variables may differ for different outputs, the cubes effective for one output may not yield good results for another. Techniques based on abstract BDDs are far superior in this case as well (See Section 3).

3 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 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 [6]). For this purpose, in Experiments 1-3 we use two abstraction functions: the *symmetric*

function and the *logarithmic* function (See Section 2).

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.

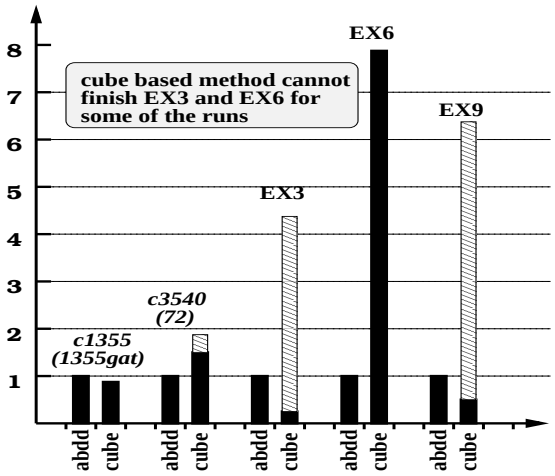

Figure 2: Static ordering with aBDD vs. cube based method

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 window based sampling gives much better results than cube based methods. 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!

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 sampling approaches, our method is also superior (Figure 3) since our method does not have the large deviation problem.

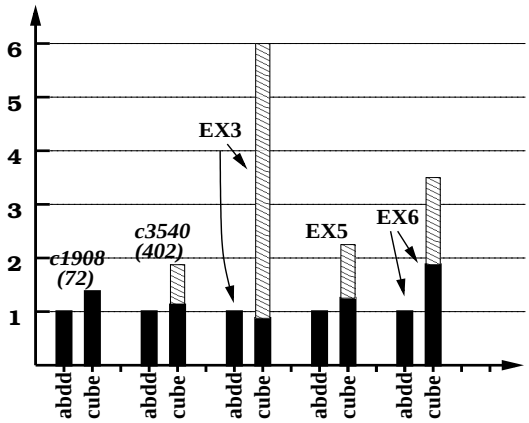

Figure 3: Dynamic ordering with aBDD vs. cube based method

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 [12] 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. Various properties dealing with handshaking, read/write transactions, and timing are checked in this model. The intial 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.

Ckts	CUDD Sift BDD Size	CUDD Sift CPU Time	Using aBDDs BDD Size	Using aBDDs CPU Time
c432	1246	0:02	1224	0:03
c499	25897	0:29	26798	1:03
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
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 for multiple output circuits

Property	# BDD Nodes SMV	# BDD Nodes Using aBDDs	TIME (sec) SMV	TIME (sec) Using aBDDs
P_1	11984K	1031K	542	246
P_2	1778K	378K	242	447
P_3	36077K	6046K	5882	1225
P_4	4458K	578K	424	198
P_5	2472K	159K	179	131
P_6	28924K	7300K	8970	3485
P_7	645K	108K	84	200
P_8	37288K	2154K	9946	835
P_9	20680K	7896K	5580	1909
P_{10}	4632K	1608K	293	198
P_{11}	19703K	4186K	2043	1008
P_{12}	38210K	2730K	2932	400
P_{13}	12740K	4252K	2831	890
P_{14}	fail	11994K	fail	2780
P_{15}	649K	100K	63	123

Table 3. Sampling for model checking

4 Summary

We have described a highly effective sampling based ordering technique. 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 also show that similar approaches using abstraction also work very well for model checking. Impressively, our technique has all of the advantages discussed previously for both static variable ordering and dynamic variable ordering.

References

[1] A. Aziz *et al.* BDD Variable Ordering for Interacting Finite State Machines. *DAC*, 1994.

[2] R. E. Bryant. Graph Based Algorithms for Boolean Function Representation. *IEEE Trans. Comp.*, 1986.

[3] E. M. Clarke *et al.* Model Checking and Abstraction. *TOPLAS*, Sept. 1994

[4] E. M. Clarke *et al.* Model Checking Using Abstract BDDs. *CMU-CS-99-124*, 1999

[5] M. Fujita *et al.* Evaluation and Improvements of Boolean Comparison Method Based on Binary Decision Diagrams. *ICCAD*, 1988.

[6] J. Jain *et al.* Sampling Schemes for Computing OBDD Variable Orderings. *ICCAD*, 1998.

[7] J. Jain *et al.* Decomposition Techniques for Efficient ROBDD Construction. *FMCAD*, 1996.

[8] S. Jha *et al.* Equivalence Checking Using Abstract BDDs. *ICCD*, 1997.

[9] Malik *et al.* Logic Verification Using Binary Decision Diagrams in a Logic Synthesis Environment. *ICCAD*, 88.

[10] K. McMillan, Symbolic Model Checking: An Approach To The State Explosion Problem. *Ph.D Thesis, CMU*, 1992.

[11] A. Narayan *et al.* Partitioned-ROBDDs - A Compact, Canonical and Efficiently Mani pulable Representation for Boolean Functions. *ICCAD*, 1996.

[12] S. Panda *et al.* Who Are the Variables in Your Neighborhood. *ICCAD*, 1995.

[13] S. Panda *et al.* Symmetry Detection and Dynamic Variable Ordering of Decision Diagrams. *ICCAD*, 1994.

[14] R. Rudell. Dynamic Variable Ordering for Ordered Binary Decision Diagrams. *ICCAD*, 1993.