X IFIP International Conference on VLSI
pp. 334-345, Lisboa, December 1999

Chapter 1

HISTORY-BASED DYNAMIC MINIMIZATION DURING BDD CONSTRUCTION

Rolf Drechsler
Wolfgang Günther
Institute of Computer Science
Albert-Ludwigs-University
79110 Freiburg im Breisgau, Germany
{ drechsle,guenther } @informatik.uni-freiburg.de

Abstract Binary Decision Diagrams (BDDs) are the state-of-the-art data structure in VLSI CAD. Since their size largely depends on the chosen variable ordering, dynamic variable reordering methods, like sifting, often have to be applied while the BDD for a given circuit is constructed. Usually sifting is called each time a given node limit is reached and it is therefore called frequently during the construction of large BDDs. Often most of the runtime is spent for sifting while the BDD is built.

In this paper we propose an approach to reduce runtime (and space requirement) during BDD construction by using history-based decision procedures. Dependent on the history of the construction process different types of sifting are called. We propose two methods that consider the quality of the hash table and the size reduction of previous sifting runs, respectively. Experimental results show that both approaches reduce the runtime significantly, i.e. by more than 40% on average.

Keywords: BDDs, Verification, Dynamic Reordering.

1. INTRODUCTION

Decision Diagrams (DDs) are often used in VLSI CAD systems for efficient representation and manipulation of Boolean functions. The most popular data structure are *ordered Binary Decision Diagrams* (BDDs) [Bryant, 1986]. Since they provide a canonical representation, for example functional equivalence of two circuits can be checked easily by building the BDDs for each circuit and then checking whether the two BDDs are isomorphic.

However, as well known BDDs are very sensitive to the variable ordering, i.e. the size of a BDD (measured in the number of nodes) may vary from linear to exponential. Finding the optimal variable ordering is an NP-hard problem [Bollig and Wegener, 1996] and the best known algorithms have exponential worst case runtime [Friedman and Supowit, 1987, Drechsler et al., 1998].

This is the reason why many authors presented heuristics for finding good variable orderings from circuit descriptions in the last few years (see e.g. [Fujii et al., 1993]). The most promising methods for BDD minimization are based on *Dynamic Variable Ordering* (DVO) [Fujita et al., 1991], i.e. improving graph size using exchanges of neighboring variables. The best results measured in the number of nodes of the resulting BDD were obtained using *sifting* [Rudell, 1993, Panda and Somenzi, 1995], but unfortunately sifting is very time consuming for large functions. Furthermore, during BDD construction it is often necessary to start sifting several times.

For this, recently several techniques for speeding up sifting have been proposed. In [Meinel and Slobodová, 1997] an algorithm has been used that partitions the search space to improve sifting runtimes. But this algorithm is largely dependent on the initial variable ordering. Another approach based on "sampling" has been suggested in [Slobodová and Meinel, 1998, Jain et al., 1998], but dependent on the chosen candidates the quality of the result varies widely, i.e. the results can be up to a factor of two worse than "classical" sifting. Lower bound sifting [Drechsler and Günther, 1999] is an approach to speed up sifting without loss of quality by computing lower bounds during variable reordering. Also a relaxed version has been proposed where a parameter has to be set by the user allowing to trade off runtime versus quality.

The focus of this paper is to show that it is not clever to use the same sifting technique all the time during BDD construction. Usually, dynamic minimization is called if a node limit is reached. Dependent on the previous results of dynamic minimization or the time elapsed since the last call of sifting, respectively, our history-driven minimization approach automatically chooses one minimization strategy. Experiments show that speed-ups of more than 40% can be observed on average.

The paper is structured as follows: In Section 2. we review basic definitions of BDDs. The sifting algorithm is discussed in Section 3. In Section 4. speed-up techniques are proposed. The new algorithms for choosing the sifting technique are introduced in Section 5. Section 6. gives experimental results. Finally, the results are summarized.

2. BINARY DECISION DIAGRAMS

As is well-known, each Boolean function $f : \mathbf{B}^n \rightarrow \mathbf{B}$ can be represented by a *Binary Decision Diagram* (BDD) [Bryant, 1986], i.e. a directed acyclic

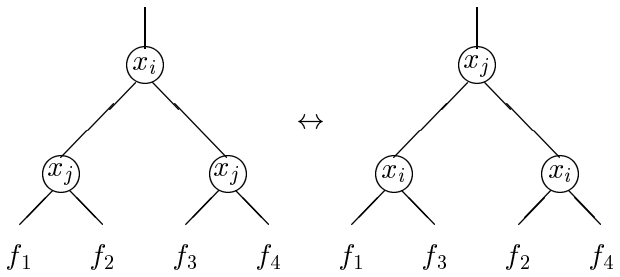

Figure 1.1 Exchange of i-th and adjacent variable

graph where a Shannon decomposition

$$f = \overline{x}_i f_{x_i=0} + x_i f_{x_i=1} \quad (1 \leq i \leq n)$$

is carried out in each node.

A BDD is called *ordered* if each variable is encountered at most once on each path from the root to a terminal node and if the variables are encountered in the same order on all such paths. A BDD is called *reduced* if it does neither contain vertices with isomorphic sub-graphs nor with both edges pointing to the same node. Reduced and ordered BDDs are canonical, i.e. for each Boolean function the BDD can be uniquely determined. Furthermore, for functions represented by reduced ordered BDDs efficient manipulations are possible [Bryant, 1986, Brace et al., 1990, Drechsler and Becker, 1998]. In the following, only reduced, ordered BDDs are considered and for briefness these graphs are called BDDs.

We briefly review an example from [Bryant, 1986] to show the importance of the variable ordering:

Example 1 Let $f = x_1x_2 + \ldots + x_{2n-1}x_{2n}$. If the variable ordering is given by $(x_1, x_2, \ldots, x_{2n})$ the size of the resulting BDD is $2n$. On the other hand if the variable ordering is chosen as $(x_1, x_{n+1}, x_2, x_{n+2}, \ldots, x_{2n})$ the size of the BDD is $\Omega(2^{n-1})$. Thus the number of nodes in the graph varies from linear to exponential depending on the variable ordering.

3. SIFTING

The basic operation of dynamic variable ordering is the exchange of adjacent variables [Fujita et al., 1991, Rudell, 1993]. The general case of exchanging a variable x_i and an adjacent variable x_j is shown in Figure 1.1. The exchange is performed very quickly since only edges must be redirected within these levels. Thus, the size is optimized without a complete reconstruction of the BDD, i.e. only local transformations for the two levels are performed, since BDDs are a canonical form.

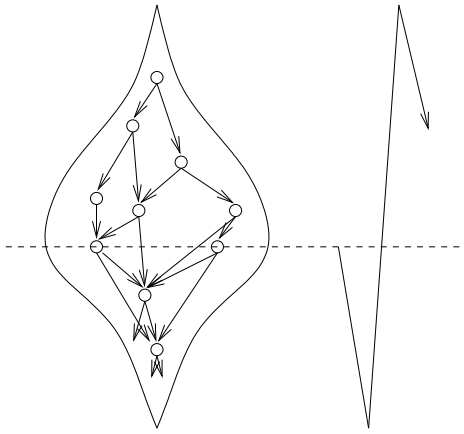

Figure 1.2 Sifting one variable

The sifting algorithm [Rudell, 1993] successively considers all variables of a given BDD. When a variable is chosen, the goal is to find the best position of the variable, assuming that the relative order of all other variables remains the same. In a first step, the order in which the variables are considered is determined. This is done by sorting the levels according to their size with largest level first. To find the best position, the variable is moved across the whole BDD. In [Rudell, 1993], this is done in three steps (see Figure 1.2):

1. The variable is exchanged with its successor variable until it is the last variable in the ordering.
2. The variable is exchanged with its predecessor until it is the topmost variable.
3. The variable is moved back to the closest position which has led to the minimal size of the BDD.

4. SPEEDING UP SIFTING

Some improvements to the original sifting algorithm have already been proposed:

Upper limit: As the size of the BDD can grow much during the movement of one variable, it is possible to set an upper limit to the growth of the BDD. If this limit is exceeded, moving into this direction is aborted. This avoids large intermediate BDDs. Using the notation from [Somenzi, 1998] in the following this "growing factor" is denoted as $maxgrowth$, i.e. $maxgrowth = 1.2$ means that the BDD may grow by at most 20% of its original size.

```
unique_find_or_add(node) {
    if (node exists) return node;
    /* a new node has to be created */
    if (next_reorder node limit is reached) {
        call sifting;
        increase next_reorder node limit;
    }
    /* some other test may follow */
    create and return new node;
}
```

Figure 1.3 Sketch of unique_find_or_add

Closest end: The considered variable is not always moved downwards first. Instead it is moved to the closest end and then to the opposite one.

Several other techniques have been proposed like the use of symmetries [Panda and Somenzi, 1995] and of interaction matrix[1] [Somenzi, 1998]. In [Drechsler and Günther, 1999] computations of lower bounds have been used to speed up sifting (so-called *lb-sifting*), i.e. often it is not necessary to move a variable to all positions. Also a *relaxed* lb-sifting has been considered. There not only a tight lower bound of $\frac{1}{2}$ is used, but a straightforward extension to $\frac{1}{b}$ ($b \geq 2$). (For more details see [Drechsler and Günther, 1999].) This extension allows to trade off runtime versus quality, i.e. the larger b, the faster the algorithm, but the resulting sizes get larger on average.

In CUDD [Somenzi, 1998], a state-of-the-art BDD package, dynamic reordering is applied if a given node limit is reached[2]. This is tested each time a new node is created. If this happens while e.g. ITE is computed [Brace et al., 1990], ITE is started again after sifting has been carried out. A sketch of the look-up technique used in ITE is shown in Figure 1.3.

Finally, for two examples it is shown how a typical BDD construction works.

Example 2 *Some data observed during BDD construction of circuits* c2670 *and* c7552 *using the CUDD package [Somenzi, 1998] is given in Figures 1.4 and 1.5, respectively. The solid line gives the relative number of nodes which are not found in the hash table during an ITE operation (see also Figure 1.3), i.e. the percentage of newly created nodes (left y-axis). The x-axis shows the number of calls of* **unique_table_find_or_add()** *during symbolic simulation. The vertical lines show where sifting is called during construction. The total memory consumption counted in number of nodes is given by the dashed line (right y-axis).*

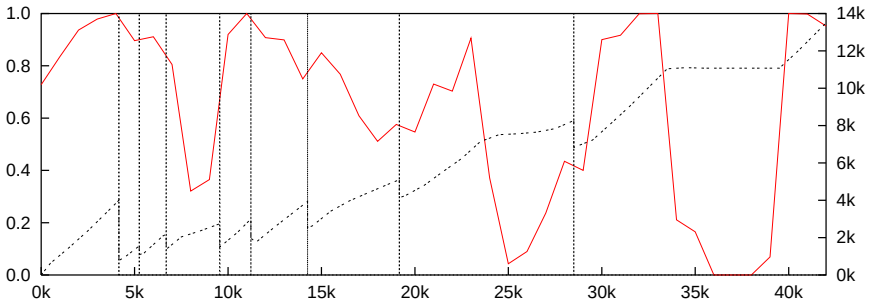

Figure 1.4 BDD construction of c2670

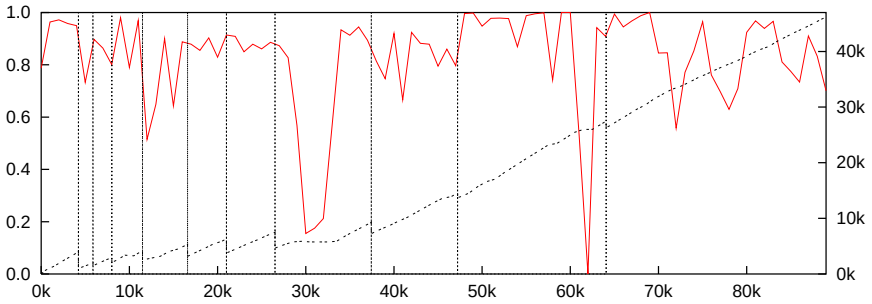

Figure 1.5 BDD construction of c7552

The following can be observed:

1. The percentage of nodes found in the unique table varies a lot and it is (more or less) randomly distributed and independent of the starting points of sifting.

2. Nevertheless, the behavior is very robust, i.e. there are phases where nodes are found all the time and phases where nodes are found very rarely. Oscillation cannot be observed.

3. At the beginning of the construction sifting gives very good reductions, i.e. often more than a factor of two, while at the end (where the BDDs are large and sifting becomes even more time consuming) the profit is marginal in many cases.

For this, calling the original sifting algorithm each time is not clever, since the algorithm is very slow, especially for large BDDs. We present in the following criteria to chose faster variants of sifting dependent on the *history* of the minimization process.

5. HISTORY-BASED DYNAMIC MINIMIZATION

In this section we present two methods for selecting a reordering heuristic that is called during BDD construction if an upper node limit is reached. For both methods we make use of relaxed lower bound sifting presented in [Drechsler and Günther, 1999]. Relaxed lb-sifting allows to dynamically trade off runtime versus quality by choosing the parameter b (see Section 4.). For sifting, we used the lower bound improvement which reduces runtime without loss of quality ("lb-sifting"). In order to distinguish lb-sifting and relaxed lb-sifting more clearly, in the following lb-sifting is also called "exact lb-sifting". Since early calls of sifting often reduce the BDD sizes much, the first call of sifting of both approaches is always exact lb-sifting.

5.1 LAST REDUCTION APPROACH

If the last call of sifting has not been very successful, it is very unlikely that a large reduction can be obtained this time. Therefore, relaxed lb-sifting can be used in that case. If, on the other side, the last reordering has reduced the BDD size much, then it is likely that this call of sifting can also reduce the BDD size, and exact lb-sifting is used. This approach prevents that too much time is spent by always applying sifting if the BDD blows-up during the construction. (For some functions the BDD size is very large independent of the variable ordering. In these cases sifting would be called frequently, vasting much runtime.)

More precisely, this can be seen as follows: if the reduction caused by sifting is small, then it is expected that also moving a single variable in the variable ordering only leads to small reductions. Therefore, the lower bound used in lb-sifting can be relaxed without much loss of quality.

To compute value b of relaxed lb-sifting, the reduction factor r of the last call of sifting is used (independently of the parameter b used in this last call). If $r \geq 0.5$, i.e. a reduction of more than 50% was observed, exact lb-sifting is used. Otherwise, b is computed by

$$1 + (maximum_b - 1)(2r - 1).$$

Thus, the smaller the reduction by the previous call, the smaller the amount of time invested in the current run. ($maximum_b$ denotes an upper limit for value b set by the user. In experiments we found that $b = 16$ is a good choice. This method is called $methodR$ (reduction) in the following.

5.2 ELAPSED TIME APPROACH

In the beginning of BDD construction, sifting is called very frequently and large size reductions can be observed. On the other hand, improvements are smaller in the long run, and sifting is called less frequently. This observation leads to the following algorithm: the parameter b is chosen dependent on the

time which has elapsed since the last call of sifting:

$$b = \text{(time since last sifting)} \times const,$$

i.e. if sifting has been called long ago, then relaxed lb-sifting is used with a large value of b, otherwise it is assumed that BDD construction has just started, and exact lb-sifting is used. For this method, "time" is measured in number of calls to **unique_find_or_add()**, i.e. the number of calls to the function which returns a node for a given function and creates a new node if necessary.

In our experiments it turned out that $const = \frac{1}{2000}$ is a good choice. We refer to this technique as $methodE$ (elapsed time) in the following.

6. EXPERIMENTAL RESULTS

In this section we describe experimental results that have been carried out on a *SUN Ultra 1* with 128 MBytes. All runtimes are given in CPU seconds. The algorithms have been integrated in the CUDD package [Somenzi, 1998]. We set a memory limit of 100 MBytes and all node counts are given in 1000 nodes. We used several benchmarks from LGSynth91 to build the BDD for and measured runtime and BDD peak node count[3]. Since results largely depend on the initial variable ordering [Harlow and Brglez, 1998], we report average values for 64 randomly chosen initial orderings. The starting value for performing sifting was 4000 nodes (which is the default value in CUDD). Then sifting was called each time the size doubled compared to the size after the last call of dynamic minimization.

To give some more insight in the experiments carried out, we first discuss the $maxgrowth$ parameter. This parameter limits the increase in size of the BDD during sifting and it turns out to be very important. For some choices of $maxgrowth$ the results are reported in Table 1.1 and 1.2. For each value the peak node count and the runtime is given, respectively. The smaller the factor is chosen the faster sifting runs, but if the value becomes too small the quality decreases significantly, and for some benchmarks the BDD cannot be built at all within the given memory limit. (Numbers in parentheses denote that the BDD could not be built for all initial variable orders.) Surprisingly, the quality did not reach its optimum for ∞.

We compared our approach to the best results obtained using lb-sifting each time, i.e. using a value of 1.2 for $maxgrowth$. For both methods we also set $maxgrowth$ to 1.2. Results are given in Table 1.3. Additionally, the average number of calls of sifting is given in column *calls*. One might expect that these numbers increase by speeding up sifting; however, this has not been observed.

As can be seen both algorithms clearly outperform sifting. Both methods significantly reduce runtime, while the peak node count is in the same range. $methodR$ even gives better results for both runtime and memory on average compared to sifting, while $methodE$ increases the peak node count by 10%,

Table 1.1 Effect of $maxgrowth$ on peak node count

circuit	#in	∞	2.0	1.3	1.2	1.1	1.0
bigtest	328	307.0	307.0	270.3	255.1	262.5	(—)
c432	36	7.9	7.9	7.6	7.5	7.3	110.6
c1355	41	153.0	153.0	146.3	141.1	136.4	328.1
c1908	33	39.4	39.4	39.3	39.3	39.1	74.1
c2670	233	32.3	32.3	28.6	27.4	28.9	(464.0)
c3540	50	292.4	292.4	233.9	225.9	208.0	1157.3
c5315	178	15.8	15.8	15.1	14.6	14.4	20.0
c7552	207	73.8	73.8	63.7	60.2	57.2	(822.8)
dalu	75	12.3	12.3	12.3	12.3	12.2	24.0
des	256	18.5	18.5	18.5	18.5	18.5	25.0
frg2	143	10.3	10.3	10.3	10.3	10.3	10.3
i8	133	16.4	16.4	16.4	16.4	16.5	31.6
i10	257	149.0	149.0	132.9	126.2	121.1	556.8
pair	173	16.1	16.1	16.1	16.1	16.0	45.5
rot	135	12.9	12.9	12.9	12.9	12.9	49.4
s1423	91	5.4	5.4	5.4	5.3	5.3	8.3
s5378	199	8.3	8.3	8.3	8.3	8.3	15.5
s9234.1	247	12.1	12.1	12.1	12.0	11.8	58.9
s13207.1	700	17.1	17.1	17.1	17.1	17.0	46.0
s15850.1	611	53.6	53.6	53.2	52.1	47.8	116.7
s38584.1	1464	51.3	51.3	50.9	49.1	46.6	(1341.4)
sum		1305.1	1305.1	1171.2	1127.7	1098.0	(5306.2)

but further reduces runtime. Compared to lb-sifting, $methodR$ is 45% faster and $methodE$ is 59% faster on average. In some cases the reduction is even more than a factor of five (see e.g. *bigtest*).

All in all, history-based selection of the reordering technique significantly speeds up the computation and often also reduces the memory needed.

7. CONCLUSIONS

In this paper history-based dynamic minimization techniques for BDD construction have been proposed. Instead of applying the same reordering algorithm all the time, dependent on the history a parameter for the reordering algorithm is determined and a faster heuristic may be used in some cases.

Table 1.2 Effect of $maxgrowth$ on CPU time

circuit	#in	∞	2.0	1.3	1.2	1.1	1.0
bigtest	328	617.0	618.1	518.2	433.7	322.2	(—)
c432	36	1.3	1.3	1.2	1.0	0.7	4.8
c1355	41	131.6	131.5	122.2	110.3	85.5	34.9
c1908	33	10.1	10.1	9.6	8.9	7.1	3.4
c2670	233	19.8	19.7	18.7	17.6	16.6	(39.8)
c3540	50	125.9	126.4	115.6	105.5	73.9	125.0
c5315	178	4.8	4.8	4.7	4.7	4.5	1.7
c7552	207	63.6	62.2	59.1	55.4	44.1	(80.5)
dalu	75	2.1	2.1	2.0	2.0	1.8	1.0
des	256	4.6	4.6	4.6	4.6	4.5	2.4
frg2	143	1.2	1.2	1.2	1.2	1.1	0.5
i8	133	2.3	2.3	2.2	2.1	1.8	1.3
i10	257	203.7	203.9	167.6	140.3	97.8	52.8
pair	173	8.9	8.9	8.9	8.8	8.2	2.1
rot	135	3.7	3.7	3.6	3.3	2.6	1.7
s1423	91	2.0	2.1	2.0	1.9	1.6	0.7
s5378	199	2.5	2.5	2.5	2.6	2.5	1.2
s9234.1	247	8.3	8.4	8.3	8.3	7.8	4.5
s13207.1	700	44.3	44.3	44.4	44.4	41.4	26.0
s15850.1	611	74.2	74.3	72.7	71.1	64.7	18.8
s38584.1	1464	213.2	212.8	212.9	212.2	208.4	(272.0)
sum		1545.2	1545.3	1382.1	1240.0	998.9	(675.0)

Experiments have shown that significant reductions of 50% in runtime on average can be observed, while the memory requirement is in the same range ($\pm 10\%$).

Notes

1. For each two variables the interaction matrix provides the information whether there is an output in the BDD which essentially depends on both variables. If two levels do not interact, the level exchange can be computed in constant time.

2. Other packages, like CAL [Ranjan and Sanghavi, 1997] from Berkeley, use different criteria. All algorithms presented in the following are also applicable in this case.

3. For sequential circuits, the transition function was used, i.e. latches were treated as additional inputs and outputs.

Table 1.3 History-based dynamic minimization for $maxgrowth = 1.2$

circuit	exact lb-sifting			$methodR$			$methodE$		
	nodes	time	calls	nodes	time	calls	nodes	time	calls
bigtest	255.1	433.7	13.3	210.4	171.3	11.8	247.9	72.4	12.4
c432	7.5	1.0	6.4	7.3	0.7	6.4	7.4	0.9	6.2
c1355	141.1	110.3	13.0	128.7	59.4	13.1	133.9	35.5	13.1
c1908	39.3	8.9	11.0	39.5	6.3	10.9	39.0	6.4	11.2
c2670	27.4	17.6	9.3	26.3	11.4	9.2	29.3	12.8	9.1
c3540	225.9	105.5	10.4	217.2	44.3	10.3	229.5	30.1	10.4
c5315	14.6	4.7	6.2	14.0	3.7	6.2	15.1	4.7	6.1
c7552	60.2	55.4	11.8	64.3	42.2	11.1	71.9	45.1	11.8
dalu	12.3	2.0	3.3	11.8	1.7	3.3	12.0	2.3	4.0
des	18.5	4.6	6.1	21.4	3.0	5.2	21.4	2.2	4.1
frg2	10.3	1.2	2.1	10.3	1.2	2.1	10.3	1.1	2.1
i8	16.4	2.1	3.9	15.9	1.7	3.9	16.0	1.8	3.5
i10	126.2	140.3	10.9	123.2	83.2	10.8	212.8	48.2	10.7
pair	16.1	8.8	12.0	15.2	6.9	11.5	14.7	7.2	11.2
rot	12.9	3.3	6.8	13.7	2.6	6.9	13.3	3.2	7.2
s1423	5.3	1.9	5.4	5.2	1.3	5.8	5.2	1.9	6.2
s5378	8.3	2.6	3.1	8.1	2.2	3.0	8.1	2.3	3.0
s9234.1	12.0	8.3	8.2	13.9	7.6	8.4	12.9	7.8	8.4
s13207.1	17.1	44.4	14.9	15.6	18.7	6.5	15.7	33.8	12.0
s15850.1	52.1	71.1	9.3	70.5	97.5	10.0	58.6	68.9	8.4
s38584.1	49.1	212.2	19.1	54.1	115.5	11.1	63.4	100.0	10.5
sum	1127.7	1240.0	186.7	1086.5	682.3	167.5	1238.3	488.5	171.5

References

[Bollig and Wegener, 1996] Bollig, B. and Wegener, I. (1996). Improving the variable ordering of OBDDs is NP-complete. *IEEE Trans. on Comp.*, 45(9):993–1002.

[Brace et al., 1990] Brace, K., Rudell, R., and Bryant, R. (1990). Efficient implementation of a BDD package. In *Design Automation Conf.*, pages 40–45.

[Bryant, 1986] Bryant, R. (1986). Graph - based algorithms for Boolean function manipulation. *IEEE Trans. on Comp.*, 35(8):677–691.

[Drechsler and Becker, 1998] Drechsler, R. and Becker, B. (1998). *Binary Decision Diagrams – Theory and Implementation*. Kluwer Academic Publishers.

[Drechsler et al., 1998] Drechsler, R., Drechsler, N., and Günther, W. (1998). Fast exact minimization of BDDs. In *Design Automation Conf.*, pages 200–205.

[Drechsler and Günther, 1999] Drechsler, R. and Günther, W. (1999). Using lower bounds during dynamic BDD minimization. In *Design Automation Conf.*, pages 29–32.

[Friedman and Supowit, 1987] Friedman, S. and Supowit, K. (1987). Finding the optimal variable ordering for binary decision diagrams. In *Design Automation Conf.*, pages 348–356.

[Fujii et al., 1993] Fujii, H., Ootomo, G., and Hori, C. (1993). Interleaving based variable ordering methods for ordered binary decision diagrams. In *Int'l Conf. on CAD*, pages 38–41.

[Fujita et al., 1991] Fujita, M., Matsunaga, Y., and Kakuda, T. (1991). On variable ordering of binary decision diagrams for the application of multi-level synthesis. In *European Conf. on Design Automation*, pages 50–54.

[Harlow and Brglez, 1998] Harlow, J. and Brglez, F. (1998). Design of experiments in BDD variable ordering: Lessons learned. In *Int'l Conf. on CAD*, pages 646–652.

[Jain et al., 1998] Jain, J., Adams, W., and Fujita, M. (1998). Sampling schemes for computing OBDD variable orderings. In *Int'l Conf. on CAD*, pages 631–638.

[Meinel and Slobodová, 1997] Meinel, C. and Slobodová, A. (1997). Speeding up variable reordering of OBDD. In *Int'l Conf. on Comp. Design*, pages 338–343.

[Panda and Somenzi, 1995] Panda, S. and Somenzi, F. (1995). Who are the variables in your neighborhood. In *Int'l Conf. on CAD*, pages 74–77.

[Ranjan and Sanghavi, 1997] Ranjan, R. and Sanghavi, J. (1997). *CAL-1.2: Breadth-first manipulation based BDD library.* University of California at Berkeley.

[Rudell, 1993] Rudell, R. (1993). Dynamic variable ordering for ordered binary decision diagrams. In *Int'l Conf. on CAD*, pages 42–47.

[Slobodová and Meinel, 1998] Slobodová, A. and Meinel, C. (1998). Sample method for minimization of OBDD. In *Int'l Workshop on Logic Synth.*, pages 311–316.

[Somenzi, 1998] Somenzi, F. (1998). *CUDD: CU Decision Diagram Package Release 2.3.0*. University of Colorado at Boulder.