Finding Compact BDDs Using Genetic Programming

Ulrich Kühne, Nicole Drechsler

Institute of Computer Science, University of Bremen, 28359 Bremen, Germany
{ulrichk, nd}@informatik.uni-bremen.de

Abstract. Binary Decision Diagrams (BDDs) can be used to design multiplexor based circuits. Unfortunately, the most commonly used kind of BDDs – ordered BDDs – has exponential size in the number of variables for many functions. In some cases, more general forms of BDDs are more compact. In constrast to the minimization of OBDDs, which is well understood, there are no heuristics for the construction of compact BDDs up to today. In this paper we show that compact BDDs can be constructed using Genetic Programming.

1 Introduction

Decision Diagrams (DDs) are used for the representation of Boolean functions in many applications of VLSI CAD, e.g. in the area of logic synthesis [1] and verification [2]. In the meantime DD-based approaches have also been integrated in commercial tools.

The state-of-the-art data structure are *Ordered Binary Decision Diagrams* (OBDDs) [2]. Since OBDDs are often not able to represent Boolean functions efficiently due to the ordering restriction [3],[4],[5] many researchers have extended the OBDD concept mainly in two directions:

1. Consider different decomposition, e.g. *ordered functional decision diagrams* (OFDDs) [6] and *ordered Kronecker functional decision diagrams* (OKFDDs) [7] make use of AND/EXOR based decompositions.
2. Loosen the ordering restriction, e.g. *general* BDDs (GBDDs) [8] allow variables to occur several times.

Following the second approach, of course, the most powerful technique is to have no restriction on the ordering at all, i.e. to use BDDs without any restrictions on ordering or variable repetition. BDDs are often exponentially more succinct than OBDDs and also for the applications mentioned above the ordering restrictions are often not needed. The main reason why OBDDs have been used more frequently is that efficient minimization procedures exist, like e.g. sifting [9]. For BDDs similar techniques are not available.

Evolutionary approaches have also been applied successfully to OBDDs, but there the problem reduces to finding a good variable ordering, i.e. a permutation

of the input variables [10]. In [11] Genetic Programming has been applied to a tree-like form of BDDs with some additional constraints.

In this paper we present an approach to BDD minimization based on Genetic Programming. In contrast to the minimization of OBDDs we carry out all operations directly on the graph structure of the BDD. By this, we present the first heuristic approach to BDD minimization. Experimental results are given to demonstrate the efficiency of the approach.

2 Preliminaries

2.1 Binary Decision Diagrams

A BDD is a directed acyclic and rooted graph $G_f = (V, E)$ representing a Boolean Function $f : \mathbb{B}^n \longrightarrow \mathbb{B}^m$. Each internal node v is labeled with a Variable $label(v) = x_i \in X_n = \{x_1, \ldots, x_n\}$, and has two successors $then(v)$ and $else(v)$. The terminal nodes are labeled with 1 or 0 corresponding to the constant Boolean functions. In each internal node the Shannon decomposition $g = x_i g|_{x_i=1} + \overline{x}_i g|_{x_i=0}$ is carried out. The m root nodes represent the respective output functions.

By restricting the structure of the BDD, special classes of BDDs can be derived:

- A BDD is *complete*, if on each path from a root to a terminal node each variable is encountered exactly once.
- A BDD is *free* (FBDD), if each variable is encountered at most once on each path from a root to a terminal node.
- A BDD is *ordered* (OBDD), if it is free and the variables appear in the same order on each path from a root to a terminal node.

OBDDs are a widely used data structure in hardware design and verification because they are a canonical representation of Boolean Functions and they provide efficient synthesis algorithms. However, for many functions the size of the OBDD depends on the variable ordering. It may vary between linear and exponential in the number of variables [2]. A lot of research has focused on the so-called variable ordering problem, which is NP-hard [12]. Furthermore, there are functions for which all variable orderings lead to an OBDD with exponential size. In turn, for some of these functions there exist FBDDs or BDDs of polynomial size [13]. This means that releasing the read-once restriction and the ordering of variables can be advantageous. But in constrast to the minimization of OBDDs by finding a good or optimal variable ordering – which is well understood [14],[9] – there are no heuristics for the construction of small BDDs up to today.

2.2 BDD Circuits

BDDs can be directly mapped to a circuit based on multiplexors. If realized with *pass transistor logic*, multiplexor cells can be used for synthesis at low cost

[15],[16],[1]. In the mapping, each internal node v of the BDD is replaced by a MUX cell. Then the 1-input (the 0-input) is connected to the MUX cell correspondig to $then(v)$ ($else(v)$). The select line is connected to the primary input $index(v)$. An example for the transformation is shown in Figure 1[1]. Obviously, the size of the BDD has direct influence on the chip area of the derived BDD circuit. For this reason, it is important to find a BDD representation as small as possible to minimize chip area.

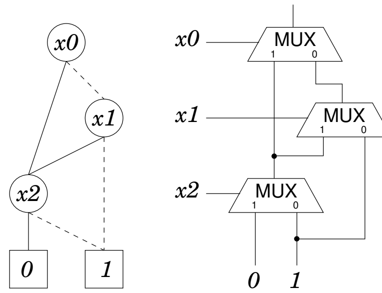

Fig. 1. Example for a BDD circuit

3 Evolutionary Algorithm

The approach presented in this paper is based on Genetic Programming (GP) [17]. Originally, GP was used to evolve LISP programs. The method at hand does not consider programs, but works directly on the graph structure of the BDDs. Several operators are provided to customize the algorithm for a specific problem. The algorithm has been implemented on top of the *evolving objects* library [18], an open source C++ library for evolutionary algorithms. The target function is kept in memory as OBDD. For this the BDD package CUDD [19] is used. The aim of the evolutionary algorithm is formulated as follows:

> The objective is to evolve BDDs that are a *correct* and *compact* representation of a given target function.

3.1 Flow of the Algorithm

The general structure of the evolutionary algorithm is based on the cycle given by the EO library. The flow is depicted in Figure 2. The algorithm is parameterized via command line switches in most of its parts. All of the operators described below can be selected and the ratios can be adjusted individually. Furthermore the algorithm provides different methods for selection, replacement and different termination criterions. This high flexibility should enable the user to customize the flow according to the respective problem.

[1] In the figures, solid lines denote *then* edges and dashed lines denote *else* edges.

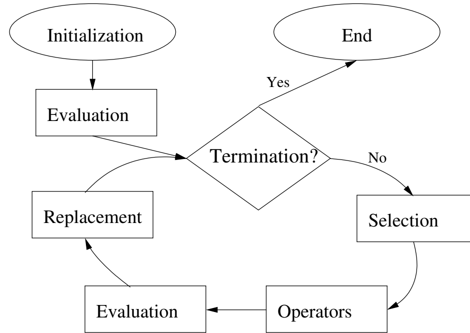

Fig. 2. The flow of the algorithm

3.2 Representation

The individuals are directed acyclic graphs with multiple root nodes, each corresponding to an output of the represented function. By adopting some popular techniques used in BDD packages, the graphs are always reduced, i.e. isomorphic subgraphs exist only once and there are no nodes with $then(v)$ and $else(v)$ being identical.

3.3 Evaluation Function

As mentioned above, there are two objectives – correctness and compactness. The two dimensions of fitness are ordered lexicographically, with correctness being the more important one. Initially there is no limitation for the structure of the represented BDDs. This means that there will be many individuals that do not represent the target function correctly. Such invalid individuals are given a worse fitness than the correct ones.

For the first dimension – correctness – it would be easy to test, if an individual represents the target function correctly or not. But this would draw no distinction between individuals that are "almost correct" and totally degenerated individuals. For this reason, a more sophisticated measure of correctness is used. The evaluation function calculates the ratio of assignments $a \in \mathbb{B}^n$ for which the function represented by the individual evaluates equivalently to the target function, i.e. the *correlation* between the individual's function and the target function. The computation of the correlation is realised by computing the XOR-BDD of the target function and the individual's function and then computing its ratio of satisfying assignments. For the latter step, the underlying BDD package provides an efficient implementation without considering each of the 2^n possible assignments.

The second dimension of fitness is the sum of the number of internal nodes of the individual and of the XOR-BDD already computed in the previous step of evaluation. The XOR-BDD is considered again in order to punish degenerated individuals that have few minterms in common with the target function. For a

correct individual, the XOR-BDD is the zero function, and only the individual's nodes are counted.

3.4 Evolutionary Operators

Table 1. Genetic operators

initialization	`Init` `CuddInit`
recombination	`NodeXover` `OutputXover`

mutation	`VariableMut`
	`EdgeSwapMut`
	`EraseNodeMut`
	`AddMinterm`

functionally conserving	`Restructuring`
	`VariableDuplication`
	`CombinedRestructuring`
	`TautoReduction`
	`SplitReduction`
	`Resize`

Table 1 gives an overview of the genetic operators. Beyond the standard classes – initialization, recombination and mutation – there are some special operators, the *functionally conserving* ones, which do not change the functional semantics of the individuals but the structure of the graphs.

Initialization. There are two types of initialization, `Init` and `CuddInit`. The first one generates random graphs with a given maximum depth. The second one creates OBDDs with a randomized variable ordering which represent the target function correctly. The name is derived from the underlying BDD package which is used for the synthesis of the OBDDs.

Crossover. `NodeXover` is basically a standard GP crossover, i.e. a node is selected from each parent, and the corresponding subgraphs are exchanged. As the individuals are rather DAGs than trees, it has to be assured that no cyclic subgraphs appear during the operation. The second crossover – `OutputXover` – performs a uniform crossover on the output nodes of the parent individuals. Thus it can only be applied to functions with multiple outputs. `OutputXover` is supposed to combine individuals that already provide good solutions for single output functions.

Mutation. Among the mutation operators there are three simple ones and the customized `AddMinterm` operator. `VariableMut` exchanges the variable of one randomly selected node. `EdgeSwapMut` selects one node and swaps its *then* and *else* egdes. `EraseNodeMut` removes one node from the graph by replacing it with one of its successors. This may be useful to eliminate redundant nodes. `AddMinterm` works as follows: first an assignment $a \in \mathbb{B}^n$ is generated. If the individual and the target function evaluate to different values under this assignment, a OBDD-like subgraph is added to the individual so that it will evaluate to the correct value afterwards. In order to create the subgraph, the target function is restricted in all variables that have been read on the path in the individual that corresponds to the assignment a. Then the new subgraph is appended to the end of this path. The operator can be used to speed up the algorithm, if the target function is relatively complex and so it would take too long to find a correct solution at all. It is a drawback that always OBDD-like subgraphs are created. This may lead to local optima that are hard to escape from.

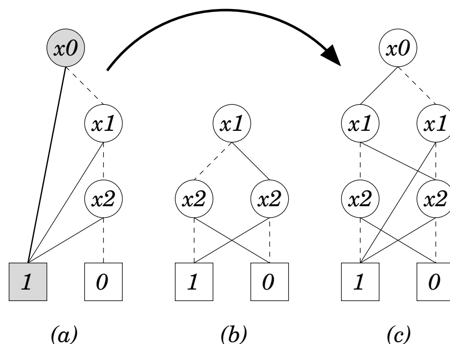

Fig. 3. Example for the `AddMinterm` operator

Example 1. Figure 3 shows an example for the `AddMinterm` operator. Let the target function be the 3 bit parity function given by $f(x_0, x_1, x_2) = x_0 \oplus x_1 \oplus x_2$. Consider the individual in Figure 3(a) and the randomly chosen assignment $a = x_0\, x_1\, \overline{x_2}$. The corresponding path is highlighted in the figure. As on this path only x_0 is evaluated, the remaining function to be added is $f_{rest} = f|_{x_0=1} = x_1 \cdot x_2 + \overline{x_1} \cdot \overline{x_2}$. The OBDD for this function (see Figure 3(b)) is added to the end of the path, obtaining the new indiviual in Figure 3(c). Note that the correlation has increased from 5/8 to 7/8, i.e. only one of eight minterms is wrong after the application of `AddMinterm`.

Functionally Conserving Operators. Among the functionally conserving operators there are two operators that perform a local restructuring of the graphs. `Restructuring` searches for subgraphs that are isomorphic to one of the graphs

shown in Figure 4 on the right and on the left. Then this subgraph is reduced to the one shown in Figure 4 in the middle. Note that the three graphs are functionally equivalent. The operator `VariableDuplication` duplicates a variable on a randomly selected path. The transformation is shown in Figure 5. In this way redundancy is added to the graph. In some cases this can lead to better solutions if the redundancy is removed elsewhere in the graph (this can be done by `TautoReduction` or `SplitReduction` which are described below). Furthermore the operator increases the diversity of the population. `CombinedRestructuring` is a combination of the two operators described above. Both are applied several times in random order.

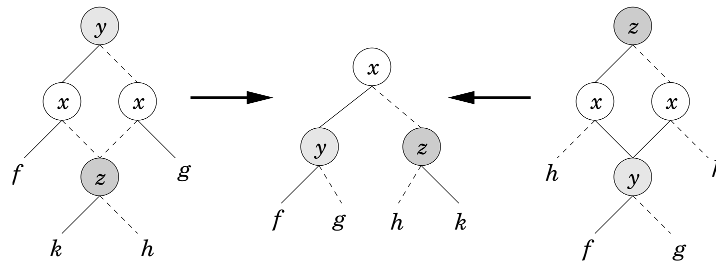

Fig. 4. The `Restructuring` operator

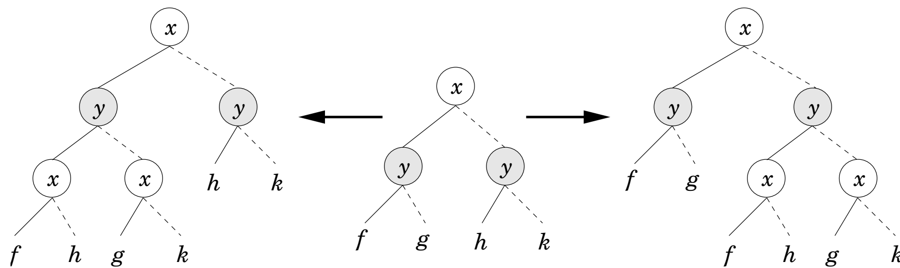

Fig. 5. The `VariableDuplication` operator

Finally there are two operators that try to reduce the number of nodes without changing the function of an individual. Algorithmically they are very similar. Figure 6 shows the `TautoReduction` operator in pseudocode. It searches for redundant nodes from top to bottom. A node v is identified redundant, if the variable $index(v)$ has already been evaluated to the same value on all paths reaching v. A redundant node can then be replaced by the appropriate successor.

Example 2. Consider the individual shown in Figure 7. The marked node is redundant because x has already been evaluated to 0 above and it can be replaced

by its *else* successor. In the next step, another node is identified redundant. In this case x has been evaluated to 1 before and the node is replaced by its *then* successor. The node marked in the third graph may not be removed because it can be reached on two paths on which x is evaluated to different values.

```
curr_lvl = TOP_LEVEL;
done = false;
while (¬done) do
    for (i = TOP_LEVEL downto curr_lvl) do
        for (each node in level i) do
            update path infos of child nodes;
        od
    od
    for (each node in level curr_lvl + 1) do
        if node is redundant then remove node;
    od
    curr_lvl = curr_lvl − 1;
    if curr_lvl ≤ 1 then done = true;
od
```

Fig. 6. Pseudocode for `TautoReduction`

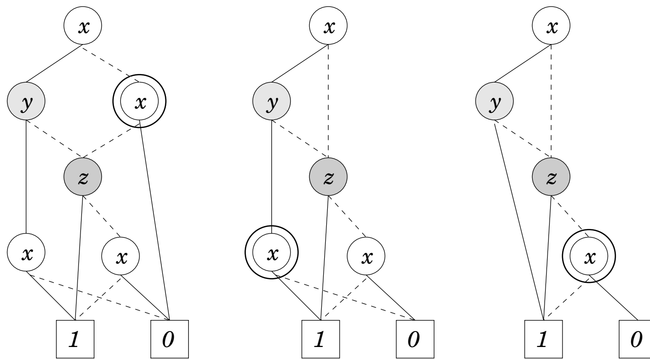

Fig. 7. Example for the `TautoReduction` operator

`SplitReduction` works similar, except that it does not remove redundant nodes but redirects edges instead. If in a node v the variable of one of its successor nodes has *always* been evaluated to a certain value and *never* to its complement, the edge to this node can be redirected to the appropriate successor node.

Example 3. Consider the individual in Figure 8. The *else* edge of the marked node can be redirected to the terminal 1-node. Furthermore the *then* edge can be redirected to the z node below because x has always been evaluated to 1 before.

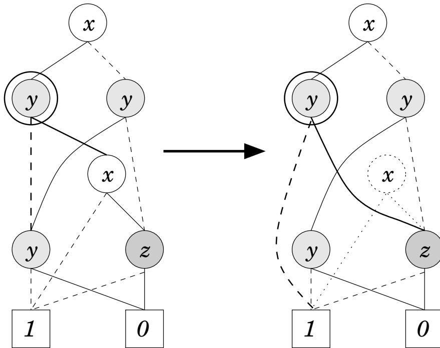

Fig. 8. Example for the `SplitReduction` operator

By redirecting edges, `SplitReduction` will possibly "relieve" a node of redundant edges and make it in turn a candidate for `TautoReduction`. It could be observed that the two operators work together quite well. Thus, there is a combination of these two operators, called `Resize`. `Resize` can be called at the end of every generation. It tries to reduce all individuals of the population using `TautoReduction` and `SplitReduction`, so that the number of nodes does not exceed a given bound. This can be used to keep the individuals relatively small in order to save run time.

4 Experimental Results

Experiments have been carried out to show the effectiveness of the approach. The runs presented in the following are supposed to exemplify the different working methods of the evolutionary algorithm. The first one starts with a population of randomly initialized graphs, the second one starts with correct OBDDs.

Example 4. Consider the *hidden weighted bit* (HWB) function given by the following equation:

$$HWB(x_0, \ldots, x_{n-1}) = \begin{cases} 0 & \text{if } w = 0 \\ x_{w-1} & \text{else} \end{cases} \quad \text{where} \quad w = |x_0, \ldots, x_{n-1}|.$$

HWB is known to have only OBDD representations of exponential size in the number of variables, but FBDDs of quadratic size [13]. In this example the presented approach is used to evolve a BDD that represents the 4 bit HWB function.

The OBDD is shown in Figure 9 on the left. The parameters are set up as follows: a deterministic tournament selection of size 2 and a comma replacement combined with weak elitism are used. The population contains 100 individuals which are initialized randomly with an initial depth of 6. The following genetic operators are applied: `OutputXover` with a rate of 0.2, `VariableMut`, `EdgeSwapMut` and `EraseNodeMut` each with a rate of 0.05 and `CombRestucturing` with a rate of 0.2. The optimal BDD shown in Figure 9 on the right could be evolved after 90 generations.

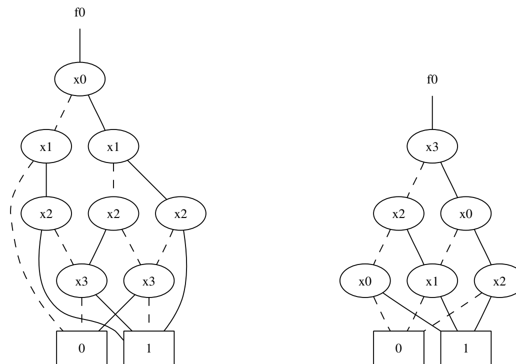

Fig. 9. OBDD and BDD for the 4 bit hidden weighted bit function

Example 5. In [2] Bryant introduced a function $f(x_0, \ldots, x_{2n-1}) = x_0 \cdot x_1 + x_2 \cdot x_3 + \cdots + x_{2n-2} + x_{2n-1}$ that is sensible to the variable ordering, i.e. the size of its OBDD varies between linear and exponential. In this example we consider a benchmark function $g : \mathbb{B}^6 \to \mathbb{B}^4$, where the four outputs are variants of Bryant's function:

$$\begin{aligned}
g_0(x_0, \ldots, x_5) &= x_0 \cdot x_1 + x_2 \cdot x_3 + x_4 \cdot x_5 \\
g_1(x_0, \ldots, x_5) &= x_0 \cdot x_2 + x_1 \cdot x_3 + x_4 \cdot x_5 \\
g_2(x_0, \ldots, x_5) &= x_0 \cdot x_3 + x_1 \cdot x_4 + x_2 \cdot x_5 \\
g_3(x_0, \ldots, x_5) &= x_0 \cdot x_5 + x_1 \cdot x_4 + x_2 \cdot x_3
\end{aligned}$$

Note that for each output function there is a different order leading to the optimal OBDD size, thus there is no global order for which all partial OBDDs are optimal. The size of the optimal shared OBDD is 31. The evolutionary approach is applied with the following settings: the initial population consists of 100 correct OBDDs. `OutputXover` and `CombRestucturing` are applied each with a rate of 0.3, `TautoReduction` and `SplitReduction` are applied with a respective rate of 0.1. Selection and replacement are the same as in Example 4. After 41 generations, an individual emerged that represents the target function with 17 nodes.

Table 2 shows additional results. Besides the name of the circuit and the numer of inputs and outputs the size of the minimal OBDD is given. The last column shows the size of the smallest BDD that could be found by our approach. The algorithm has been run 50 times with a limit of 200 generations and a population size of 100. Only for the largest benchmarks with 7 inputs a population size of 200 and a generation limit of 300 have been used. As can be seen, in many cases smaller representations could be found. Especially the HWB and ISA functions for which there is an exponential gap between their OBDD- and BDD-size show good results. But also for randomly generated functions the graph size could be improved. For other benchmarks no improvements could be made, but it should be noted that for numerous common functions there are OBDDs of linear size and thus no improvements can be expected by using BDDs instead.

Table 2. Benchmark circuits

circuit	i/o	OBDD	GA	circuit	i/o	OBDD	GA
rnd_3_3_a	3/3	9	7	hwb4	4/1	7	6
rnd_3_3_b	3/3	7	7	hwb5	5/1	14	12
rnd_3_3_c	3/3	8	7	hwb6	6/1	21	20
rnd_3_3_d	3/3	9	8	isa2	5/1	10	8
rnd_4_2_a	4/2	10	11	isa3	10/1	26	20
rnd_4_2_b	4/2	12	11	f51m.49	3/1	4	4
rnd_4_2_c	4/2	11	12	cm82a.f	3/1	5	5
rnd_4_2_d	4/2	12	11	b1	3/4	8	8
rnd_5_1_a	5/1	11	11	f51m.48	4/1	6	6
rnd_5_1_b	5/1	11	11	cm42a.e,f	4/2	5	5
rnd_5_1_c	5/1	12	11	cu.pq	4/2	10	10
rnd_5_1_d	5/1	12	12	cm82a.h	5/1	7	7
rnd_5_4_a	5/4	37	35	C17	5/2	7	7
rnd_5_4_b	5/4	35	34	cm138a.m	6/1	6	6
rnd_5_4_c	5/4	34	32	pcl3.3	7/1	8	8
rnd_5_4_d	5/4	39	38	cu.rs	7/2	8	8

5 Conclusions and Future Work

In this paper it has been shown that it is possible to construct compact BDDs using genetic programming. First experiments have yielded some promising results. However, there are still some problems to be solved. For large functions that depend on many variables, it takes too long to evolve a correct solution from a randomly initialized population. This can be avoided, if correct OBDDs are used as initial population. Unfortunately, the regular structure of the OBDDs seems to be very stable, and the algorithm will hardly escape from the local optima induced by the OBDDs. Certainly there is still capability for improvements. Possibly new operators that act less locally than `Restructuring` could help to "break" the OBDDs.

References

1. Drechsler, R., Günther, W.: Towards One-Path Synthesis. Kluwer Academic Publishers (2002)
2. Bryant, R.: Graph-based algorithms for Boolean function manipulation. IEEE Trans. on Comp. **35** (1986) 677–691
3. Ajtai, M., Babai, L., Hajnal, P., Komlos, J., Pudlak, P., Rödl, V., Szemeredi, E., Turan, G.: Two lower bounds for branching programs. In: Symp. on Theory of Computing. (1986) 30–38
4. Bryant, R.: On the complexity of VLSI implementations and graph representations of Boolean functions with application to integer multiplication. IEEE Trans. on Comp. **40** (1991) 205–213
5. Becker, B., Drechsler, R., Werchner, R.: On the relation between BDDs and FDDs. Technical Report 12/93, Universität Frankfurt, 12/93, Fachbereich Informatik (1993)
6. Kebschull, U., Schubert, E., Rosenstiel, W.: Multilevel logic synthesis based on functional decision diagrams. In: European Conf. on Design Automation. (1992) 43–47
7. Drechsler, R., Sarabi, A., Theobald, M., Becker, B., Perkowski, M.: Efficient representation and manipulation of switching functions based on ordered Kronecker functional decision diagrams. Technical Report 14/93, J.W.Goethe-University, Frankfurt (1993)
8. Ashar, P., Ghosh, A., Devadas, S., Newton, A.: Combinational and sequential logic verification using general binary decision diagrams. In: Int'l Workshop on Logic Synth. (1991)
9. R.Rudell: Dynamic variable ordering for ordered binary decision diagrams. In: Int'l Workshop on Logic Synth. (1993) 3a–1–3a–12
10. Drechsler, R., Becker, B., Göckel, N.: A genetic algorithm for variable ordering of OBDDs. IEE Proceedings **143** (1996) 364–368
11. Sakanashi, H., Higuchi, T., Iba, H., Kakazu, Y.: Evolution of binary decision diagrams for digital circuit design using genetic programming. In: ICES. (1996) 470–481
12. Bollig, B., Wegener, I.: Improving the variable ordering of OBDDs is NP-complete. IEEE Trans. on Comp. **45** (1996) 993–1002
13. Wegener, I.: Bdds – design, analysis, complexity, and applications. Discrete Applied Mathematics **138** (2004) 229–251
14. Friedman, S., Supowit, K.: Finding the optimal variable ordering for binary decision diagrams. In: Design Automation Conf. (1987) 348–356
15. Buch, P., Narayan, A., Newton, A., Sangiovanni-Vincentelli, A.: On synthesizing pass transistor networks. In: Int'l Workshop on Logic Synth. (1997)
16. Ferrandi, F., Macii, A., Macii, E., Poncino, M., Scarsi, R., Somenzi, F.: Layout-oriented synthesis of PTL circuits based on BDDs. In: Int'l Workshop on Logic Synth. (1998) 514–519
17. Koza, J.: Genetic Programming - On the Programming of Computers by means of Natural Selection. MIT Press (1992)
18. M. Keijzer, J.J. Merelo, G.R., Schoenauer, M.: Evolving objects: a general purpose evolutionary computation library. In: Evolution Artificielle. (2001)
19. Somenzi, F.: CUDD: CU Decision Diagram Package Release 2.4.0. University of Colorado at Boulder (2004)