"Vasile Alecsandri" University of Bacău
Faculty of Sciences
Scientific Studies and Research
Series Mathematics and Informatics
Vol. 19 (2009), No. 2, 241 - 256

NEW HYBRID GENETIC ALGORITHM WITH ADAPTIVE OPERATORS AND VARIABILITY TARGET FOR OPTIMIZING VARIABLE ORDER IN OBDD

I. FURDU AND O. BRUDARU

Abstract. Reduced Ordered Binary Decision Diagrams are one of the most powerful data structure for boolean manipulation on switching functions as top process in digital circuits design. The size of ROBDDs is very sensitive to the ordering choices of input variables. A new genetic algorithm is described for optimizing the variable order. It uses adaptive operators and includes a mechanism based on information energy for controlling the variability of the population. Experimental investigations of the performance of this genetic algorithm are described.

1. INTRODUCTION

The OBDD size is given by the number of its nonterminal nodes. A smaller number of nodes imply a smaller circuit design, whereas a bad ordering can lead to an exponential growth in the size of OBDD.

The existing variable ordering methods [10, 12, 18] include static variable ordering techniques and dynamic variable ordering techniques. Most of static techniques are used to determine good initial variable orders before constructing the OBDD of a function and they are usually based on a breadth-first or depth-first traversal of a circuit [11, 13] from its outputs to inputs. Dynamic techniques are based on a process of improving the variable order and the size of an already built OBDD [4, 19, 22]. The Rudell's sifting algorithm and window permutation algorithm [21, 24] are the most popular dynamic techniques.

In experimental studies, it turned out that methods based on genetic algorithms yield better results than other techniques. Even the runtime for genetic reordering algorithms were brought down to a reasonable level, they are still not competitive to deterministic reordering heuristics like window permutation or sifting [16].

Keywords and phrases: BDD, OBDD Optimisation, GA
(2000) Mathematics Subject Classification: 06E30, 94C10, 68W35

Genetic algorithms in finding the best ordering were first introduced in [7] where the main genetic operations are PMX (partially-mapped crossover) and mutation. Simulated annealing used in [1] is a related approach. These two methods yield better results than other techniques, but they are comparably slow (see e.g.[24]) as any other GA based method. In order to speed up the computations, other approaches have been suggested [8, 23] which include treating of sifting as a genetic operation that replaces crossover techniques or advanced tricks for setting the parameters.

Recent research [5, 6, 9] have shown that parallel or distributed genetic algorithms are a feasible way to solve the problem of the best ordering, but the cost of fitness evaluation remain an expensive task which affects the global cost of solution even such algorithms benefits from asynchronous behavior or from the independence of the processes.

This paper focuses on the purely part of genetic reordering algorithms in order to improve the solutions quality. The proposed GA uses three mutation operators and three crossover operators that are applied with given probability distribution. The sifting heuristic method [22] is grafted on the GA, being embedded in a so-called hypermutation operator. Another contribution is a formal framework to increase the quality of solutions by using adaptive operators and a mechanism based on information energy [20] for controlling the population variability. Experimental results prove that using a strategy to modify the probabilities of genetic operators according to prescribed variability policy and adapting the amount of changes produced by these operators to evaluation stages improve OBDD size. Section II treats the importance of variable ordering in OBDDs, section III describes the proposed hybrid genetic algorithm and the formal framework for adapting operators and their application to increase the solutions quality. Section IV presents the experimental results and last section summarizes the work.

2. THE IMPORTANCE OF VARIABLE ORDERING IN OBDD

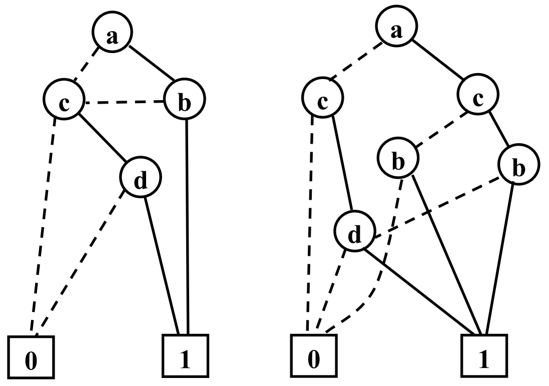
OBDDs have numerous applications- e.g. in formal verification of digital circuits and other finite state systems- have been found and manipulation algorithms or BDD derived data types have improved time and memory performance [4].

Let $f : B^n \to B^m, B = \{0,1\}$ be a switching function and *л*- a total order on a fixed set of boolean variables $x_1, x_2, ..., x_n$. An OBDD with respect to order л is a single rooted direct acyclic graph that satisfies the following properties [3]:

a) there are exactly two terminal nodes labeled by boolean constants 0 and 1, respectively.
b) each non-terminal node is labeled by a variable x_i, and has two outgoing edges, called 0-edge and 1-edge. In each inner node, two subfunctions are usually computed according to Shannon decomposition [18].
c) the order in which the variables appear on a path in the graph is consistent with the variable order л. d) further reduction rules could be applied [3] to obtain a canonical representation for *f*, a so-called reduced (R)OBDD. Usually, the widely used term OBDD (or simply BDD) refers to a ROBDD.

In the problem of optimizing the variable order in OBDD's, a boolean function *f* which describes a digital circuit is represented as a reduced OBDD.

OBDDs share some bad properties with all kinds of switching functions representations: the size of OBDDs strongly depends on the order of input variables (figure 1) and can vary from linear to exponential. The process of improving the variable ordering in OBDDs is a NP- complete problem [2].



(a) 4 nodes (b) 6 nodes

Fig. 1. *The influence of the variable ordering for f(a,b,c,d)=a·b·c·d with order a, b, c, d (a) and order a, c, b, d (b); 0- edges are dashed.*

Another critical aspect is related to large amount of memory or computing time consumption needed by the algorithm for optimizing the circuit in case of certain complex functions. There are a variety of methods to find the optimal variable ordering for BDDs but none can fulfill both time and space requirements of the circuit.

The variable reordering problem of OBDDs is a typical combinatorial problem with a huge search space of possible solutions. If an OBDD for a function *f* and a variable ordering л is needed, we don't have an efficient procedure to compute the size estimation of OBDD for a certain *f*, so we have to construct the OBDD. This can be efficiently done merely in the case when the size of the resulting OBDD is polynomially related to the size of the initial OBDD [19]. Hereby, heuristics that choose several candidates for a “good” ordering cannot avoid the construction of the OBDDs for their evaluation.

Fortunately, for most functions in real-life applications, we can found a variable order that keeps the size of the corresponding OBDD tractable. Hence, for most practical applications, OBDD are efficient for manipulating switching functions.

In this approach, six genetic operators are used: three for mutation and three for crossover, which include a variant of the alternative crossover [16]. In addition, a GA’s hybridization technique is proposed by using a partial application of sifting [22] as a regular hypermutation operator. Adaptive operators and a strategy to adjust the probabilities of genetic operators according to a variability target in order to improve OBDD size are proposed.

3. A GENETIC ALGORITHM FOR THE VARIABLE ORDERING PROBLEM

Further, the main components of the proposed GA are summarized.

The GA’s main directions of improvement the OBDD size are: (i) more operators are applied with given probability distribution, three for mutation and three for crossover. (ii) sifting technique is grafted on the GA by a hypermutation operator and (iii) the use of a control mechanism based on information energy [20] in order to adapt the population variability.

III.1. **Solution representation.** Each individual represents a specific OBDD variable order in permutation form [15] assuming the initial variables input order is the natural one. Every gene represents one input variable by an integer value in range [1, n], without duplicates, according to the position of the variable in the stated order.

III.2. **Initial population.** Initial population *P* is randomly generated. Additionally, the individual obtained by applying sifting for identical permutation *1, 2, ...n* is also included. Population size is empirically indicated in table 1.

n	<20	21-200	201-300	301-400	>400
Popsize	50	60	75	90	120

Table 1. Population size.

Survival selection. The selection is deterministic and elitist [14]. At the end of each stage current population competes with the new individuals and those with better fitness survives in the limit of $|P|$ for the next stage.

Stop condition. Algorithm stops if the variation of the ratio $[\phi_m(k)-\phi_m(k-1)]/\phi_m(k)<tol$, where $\phi_m(k)$ is the average of the fitnesses

at stage *k*, *tol* is a prescribed tolerance, and no further improvement for average fitness is observed for a fixed n_{add} number of iterations.

III.3. **Fitness function.** The fitness function computes for each chromosome its number of nodes in the corresponding OBDD. Fitness is computed by using *Nanotrav* tool included in CUDD package [24].

III.4. **Mutation.**

Operators. Three mutation operators are used: simple mutation (mutual exchange which means the exchange of the positions of two randomly selected genes), group mutation- a group of genes is moved from one position to another (group length, first and last position- chosen randomly) and inversion- in which two cutpoints are randomly selected and the ordering in the enclosed segment is reversed.

Adaptive operators. As the algorithm converges, the disruption needs to be small in order to preserve good schemata. In order to apply less disruptive mutation we have to control the cutting segment length for each mutation type. A big enough cutting segment gives a more different individual, which is to prefer in early stages of the algorithm. Thus, the length of the cutting segment is controled by a linear decreasing function: it decreases from 1/3 of the input length at the first iteration to 1 when the number of iterations riched *it_max* (1).

$$f(t) = at + b, f(0) = nrvar/3, f(it\_max) = 1 . \tag{1}$$

where *it_max* is an estimation of maximum number of iterations and *nrvar* = input length = chromosome length.

Selection and application. A randomly chosen individual from population bears mutation with probability p_m. If a mutation operation is decided, one of the three mutation operators is randomly selected according to a given probability distribution. This probability distribution remains fixed over the whole evolution process.

III.5. **Crossover.**

Operators. The algorithm uses three crossover operators. *Partially matched crossover* (PMX) [14] selects a matching section between two cutpoints and uses exchange operations to make first parent's matching section assimilate the second ones. The second is *order crossover* (OX) in which every element between two randomly selected cutpoints is copied from the first parent, and the elements outside the cutpoints are filled with the missing genes from the other parent, preserving its order [14]. The third crossover operator is a variant of *alternating crossover* [16] but instead of taking one gene alternately from each parent (more disruptive), a group of genes is used.

Adaptive operators. In order to encourage the growth of the constructive blocks, their formation and preservation as the evolution advances, the distance between the cutting points is adapted to the number of iterations. The length of the cutting segment (matching section for PMX) for all crossover types is controled by a linear decreasing function. This length also, will decrease with the number of generations from one third of the chromosome length at the beginning to 1 when *it_max* is reached (1).

Selection and application. The matting pool is given by the first 50% most efficient individuals. One individual is chosen for crossover with probability p_c. Its mate is randomly chosen also from the matting pool. For each pair of parents one of the three crossover operators is applied according to a given distribution probability which remains constant over the entire evolution.

III.6. **Hypermutation.** The main function of hypermutation is to graft sifting heuristics on the GA. One arbitrary chosen individual from population bears hypermutation with probability p_H which remains constant during the evolution process. When hypermutation is applied to an individual, two randomly cutpoints that forms a segment are chosen. Swap steps are restricted to the genes within the segment, the best position that gives the minimum number of nodes for one arbitrary chosen gene is keeped, and a new individual is generated. An adaptive mechanism is applied for hypermutation, too. The length of the segment is also adapted to the number of iterations: it is equal to 15% of the chromosome length at iteration 1 and decreases up to 5% when *it_max* is reached.

III.7. **Tuning the genetic operator's application.**

Another mechanism to improve performance of the proposed GA is to use a variability policy in order to find the best strategy for applying genetic operators. The variability of population generally decreases as the algorithm converges and the interval between the best and the worst fitness shrinks.

Let P be the current population of $x_1, x_2, \ldots x_m$ chromosomes at the beginning of a given evolution stage. For each chromosome its fitness $fit(x_i)$, $i = 1, \ldots m$, is computed.

Since the first iteration, best and worst fitness are computed for each population, after evaluation step: $fm = \min\{fit(x_i) / i = 1, \ldots m\}$ and $f_M = \max\{fit(x_i) / i = 1, \ldots m\}$.

The interval $[f_m, f_M]$ is divided in p equal lenghts subintervals I_k, where $k=1, \ldots p$, $p \approx |P|/10$.

Let F_k be the number of fitness values which belong to I_k, $k=1 \dots p$ and $F'_k = F_k / |P|$. Obviously, $\sum_{k=1}^{p} F'_k = 1$.

The "energy" of the current population [20] is defined as:

$$E_c = \sqrt{\sum_{k=1}^{p} (F_k')^2}\ . \tag{2}$$

It gives a measure of population variability. It holds: $1/\sqrt{P} \le E_c \le 1$.

Thus, a small E_c indicates a high variability and this should be associated with the beginning of the algorithm evolution and a high E_c indicates a small variability recommended being at the end of the run.

In order to find an appropriate policy to vary the probabilities for operators application four types of variability target functions E_{ob} are considered, each depending on the number of iterations t (except first):

a.constant:

$$E_{ob} = B \in [1/\sqrt{p}, 1]. \tag{3}$$

b. linear increasing:

$$E_{ob}(t) = at + b, E_{ob}(t) = 1/\sqrt{p}, f(it\_\max) = 1. \tag{4}$$

c. periodic:

$$E_{ob} = (A_1 - A_2)\,|\cos \omega t| + A_2,\quad 1/\sqrt{p} \le A_2 < A_1 \le 1,\quad \omega = 2,3\dots. \tag{5}$$

d. exponential periodic:

$$E_{ob}(t) = e^{-\lambda / t}\,|\cos \omega t|, \text{with } \lambda = 1/2 \ln p\ . \tag{6}$$

Let $\lambda = (E_{ob} - E_c)/T$ be an adjustment parameter, where T= 2, 3, ... is also a parameter. Parameter λ adjusts the speed wherewith the target value E_{ob} is reached by the current value of the information energy E_c.

The probabilities for mutation and crossover are adjusted using the equations:

$$\begin{aligned} p_m &= p_m - \alpha\lambda \\ p_c &= p_c + \beta\lambda, \end{aligned} \tag{7}$$

where $\alpha, \beta \in [0,1]$. Those p_m and p_c values which exceed 0 or 1 are forced to 0 or 1, respectively.

Consider E_{ob} a generic target of variability. The way to adapt p_m and p_c is given by the rules (7): remark that if current energy $E_c < E_{ob}$, then $\lambda>0$ so p_c is growing and p_m is decreasing in order to lower the variability. If current energy $E_c > E_{ob}$, then $\lambda<0$ and, as a result, a lower p_c and a higher p_m are obtained, which will increase variability.

4. PERFORMANCE EVALUATION

In experiments a subset of LGSynth91 benchmarks is used, obtainable from [25]. For OBDD manipulation the package used was CUDD [24]. The experiments were conducted on a Dual Core system, with 2,4 GHz processors, 2G RAM available memory and Linux. Number of runs per circuit test was 10 for each type of experiment.

IV.1. **Setting of parameters.** In the first series of experiments, various population sizes, stop criteria, parameters values were tested in order to find adequate values. Consequently, population size is indicated in table 1. Adequates distribution of probability for PMX, AX, OX are 0.2, 0.4 and 0.6 respectively and for simple mutation, inversion and group mutation are 0.2, 0.3 and 0.5, respectively. Hypermutation is applied with probability p_H= 0.1 and is not controlled by the target variability mechanism. A recommended value for *tol* is 0.001 and for n_{add} is 20. Other recommended values for parameters: γ=1 for E_{ob} constant; A_1=0.9, A_2 =0.45 for E_{ob} periodic, ω=3 and T=4 for E_{ob} periodic and E_{ob} exponential periodic.

IV.2. **Performance estimation.** The second series of experiments were conducted in order to find the best policy for variability target. Table 2 shows the corresponding values for best ever fitness reported *F_best* [22, 24] in literature, followed by the input and output number of nodes for each benchmark. The minimum number of nodes obtained for each benchmark is given in column *F_min* and the variability target policy that produced it in column *F_type*.

Name	***F_best***	***In***	***Out***	***F_min***	***F_type***	***F_min -F_best***	***(F_min-F_best)/F_best***
apex6	498	135	99	589	periodic	91	0.182731
C499	25866	41	32	26153	exp_per	287	0.011096
vda	478	17	39	478	exp_per	0	0
misex3	478	14	14	478	periodic	0	0
dalu	689	75	16	714	periodic	25	0.036284
cordic	42	23	2	42	constant	0	0
ttt2	107	24	21	107	periodic	0	0

Table 2. Best variability policy.

Absolute and relative fitness errors are also given. For *ttt2*, *misex3* and *cordic* benchmarks, where the same minimum was obtained for more variability target policies, the best policy was chosen according to the number of *F_min* occurencies (not shown here). The obtained values seem to prove that the best variability policy is the periodic (5) one. Certainly, constant and linear target variability policies do not improve the performances significantly by comparing with periodic and exponential periodic policies. Periodic variability target seems to give slightly better results then the exponential periodic variability target. A comparison between linear and periodic variability target policies for benchmark *apex6* is illustrated in figure 2. (*avg_F* = average of fitnesses).

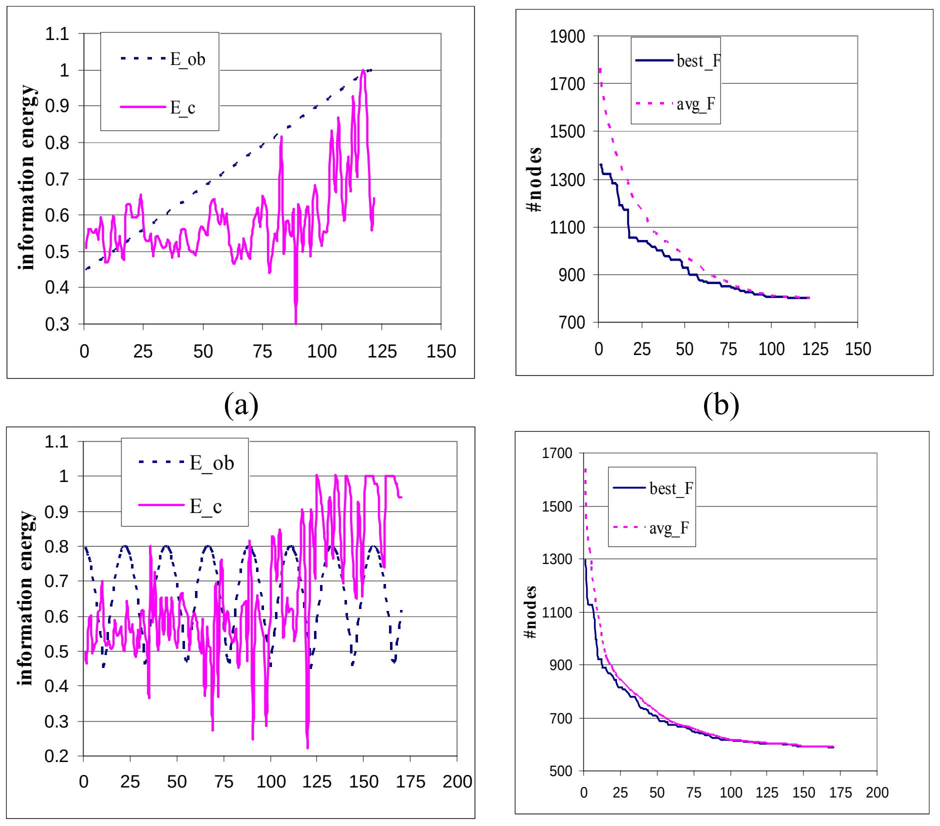


Fig. 2. Comparison between E_{ob} linear ((a)-(b)) and E_{ob} periodic ((c)-(d)) for *apex6* benchmark.

Table 3 gives the average, standard deviation and unitized risk for absolute error distribution *F_min-F_best* and for each type of target variability. In order to observe which policy performs better, in table 4 same data for relative frequencies (*F_min-F_best)/F_best* are given.

Name		***apex6***	***C499***	***vda***	***misex3***	***dalu***	***cordic***	***ttt2***
constant	avg	160.0000	3930.6999	2.8000	13.9000	217.6999	2.1000	3.2000
	σ	27.3464	3730.7937	0.9742	20.3597	36.4336	2.6089	2.2876
	σ/avg	0.1709	0.9491	0.7062	1.4647	0.1673	1.2423	0.7148
linear	avg	179.5999	2772.7001	5.3000	16.2000	173.8999	4.4000	1.5000
	σ	51.9529	2620.1159	4.4988	19.7414	83.1630	4.8620	1.4317
	σ/avg	0.2892	0.9449	0.8488	1.2186	0.4782	1.1050	0.9544
periodic	avg	123.3000	2681.8999	2.4000	11.7000	161.7000	3.9000	2.5
	σ	25.3566	2519.5214	1.2806	17.9451	65.1408	4.6804	2.5446
	σ/avg	0.2056	0.9394	0.5335	1.5337	0.4028	1.2001	1.0178
exp_periodic	avg	134.6000	2205.1000	1.9000	15.2000	145.6000	3.8000	3.1000
	σ	20.2593	1737.8807	1.8676	19.0812	63.0957	3.6848	2.5735
	σ/avg	0.1505	0.7881	0.9829	1.2553	0.4333	0.9696	0.8301

Table 3. Average, standard deviation and unitized risk of *F_min-F_best* for each variability target

Name		***apex6***	***C499***	***vda***	***misex3***	***dalu***	***cordic***	***ttt2***
constant	avg	0.3212	0.1033	0.0058	0.0290	0.3103	0.0214	0.0299
	σ	0.05491	0.1522	0.0029	0.0425	0.0430	0.0213	0.0206
	σ/avg	0.1709	1.47296	0.5101	1.4647	0.1387	0.9979	0.6912
linear	avg	0.3606	0.1265	0.0110	0.0338	0.2523	0.0119	0.0214
	σ	0.1043	0.1089	0.0094	0.0413	0.1207	0.14263	0.0118
	σ/avg	0.2892	0.8610	0.8488	1.2186	0.4782	11.9816	0.5516
periodic	avg	0.2160	0.1398	0.0050	0.0244	0.2346	0.1357	0.0233
	σ	0.0512	0.1254	0.0026	0.0375	0.0945	0.1158	0.0237
	σ/avg	0.2374	0.8967	0.5335	1.5337	0.4028	0.8532	1.0178
exp_periodic	avg	0.2702	0.0852	0.0054	0.0014	0.1864	0.1843	0.0289
	σ	0.0228	0.0671	0.0039	0.0386	0.0961	0.1224	0.0240
	σ/avg	0.0844	0.7881	0.7256	1.2150	0.5155	0.6643	0.8301

Table 4. Average, standard deviation and unitized risk of (*F_min-F_best)/ F_best* for each variability target

Table 4 confirms that the periodic and exponential periodic policies are better than the others. For the first two benchmarks values in table 3 and 4 fits *F_type* from table 2; for *vda* periodic variability policy has a slight advantage over exponential variability policy. Instead, for the last benchmark, linear policy wins as table 3 shows. Overall, a periodic variability target is the best candidate in order to optimize genetic operator's effect. In the following only this policy is used.

Further, the influence of periodic function parameters A_1, A_2 (5) is studied. Three sets of 10 runs are made (*apex*). Because periodic function varies between A_2 and A_1, in the first set of runs we lower the upper limit, in the second we rise the inferior and in the last both to the average distance between them. Table 5 shows that there are no further improvements for *F_min*, hence, the distance between A_1, A_2 should be kept at maximum.

A1, A2	*A1=0.6, A2=0.45*	*A1=0.9, A2=0.6*	*A1=0.55, A2=0.7*
F_min	599	612	607

Table 5. Setting parameters for periodic function

In table 6 is presented for each benchmark, for periodic variability target, average number of iterations until algorithm stop *it_stop*, average number of fitness evaluations *avgF*. In order to evaluate the cost of policy application, the given values includes the benchmarks for which periodic variability target was not the best one.

Name	***Periodic***	
	it_stop	*avgF*
apex6	182.2	4212.7
C499	122.6	3310.6
vda	62.1	1644.7
misex3	61.6	1623.4
dalu	148.1	3250.0
cordic	89.2	2147.3
ttt2	71.7	1672.7

Table 6. Costs for periodic variability target

Costs are also related to structural benchmark complexity in terms of computational time.

Figure 3 illustrates the algorithm behavior with periodic variability target and adaptive operators for one benchmark (*dalu*) run. It can be observed that the current energy (2) of the algorithm cannot follow with accuracy the objective energy. The process of current energy adaptation to objective energy has, however, a normal oscillatory trend. When current energy is down, a lower amount of mutation is compensated by a higher rate of crossover operations (7). Lower peaks for current energy observed around iterations 100, 120 are because, for some period, mutation probability is 1. At the end of the run a high value for current energy fits the idea that population variability is low. Figure 4 presents for this runtest the best *best_F* and the average fitness *avg_F* according to the number of generations. A very good convergence can be observed.

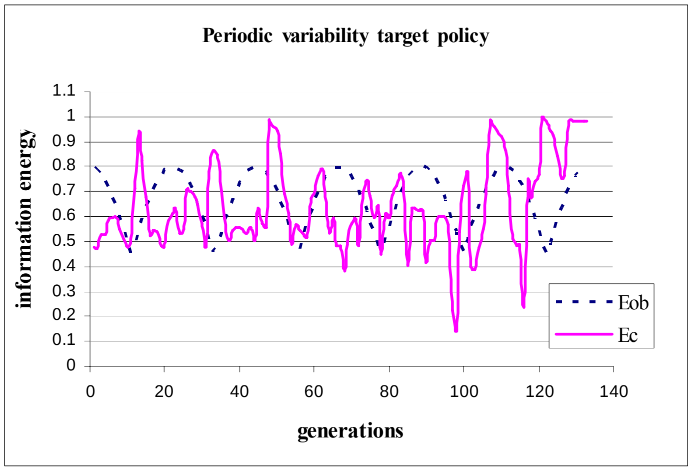


Fig. 3. Example of periodic variability target behavior (*dalu benchmark*).

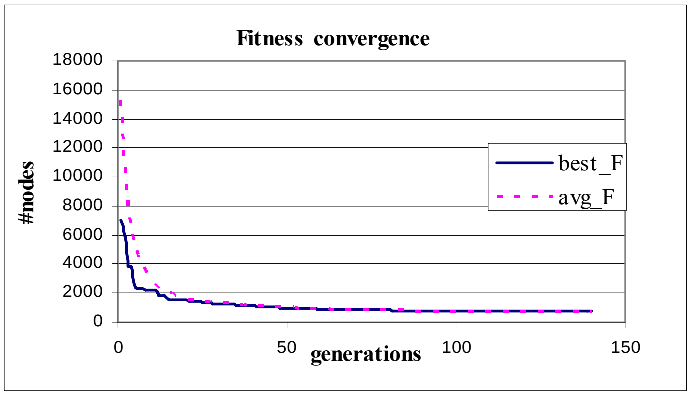


Fig. 4. Best fitness and average fitness convergence for *dalu* benchmark.

IV.3. **The effect of hypermutation.** Third set of experiments studies the effect of hypermutation within periodic variability target policy. *Apex6* benchmark was selected for tests, as it gives the worst result compared to *F_best*. Hypermutation was applied with different probabilities p_H. Average time, minimum number of nodes *F_min* obtained, the average, standard deviation and unitized risk for *F_min-F_best* are presented in table 7. High values of hypermutation probability lead to better fitnesses. Thus, a good policy is to apply hypermutation with a high rate. Unfortunately, higher p_H values lead to a growing computational time as second row in table 7 indicates. Tests on other benchmarks confirm this result. In order to observe the postoptimization effect of hypermutation, we applied hypermutation at the end of the runs, for the first best five chromosomes. Half of their genes were shifted within a window length equal to the chromosomes length.

p_H	0.05	0.1	0.15	0.2	0.25
time	100.240	103.298	131.664	137.405	145.824
F_min	592	589	574	569	564
F_min - F_best	94	91	76	71	66
avg.	152.5	152.1999	134.3000	128.5000	123.3000
σ	34.3203	31.7000	37.6269	38.2057	25.3566
σ/avg	0.2250	0.2082	0.2801	0.2973	0.2056

Table 7. The effect of hypermutation for *apex6* circuit with periodic variability target policy.

No further improvement was obtained except a single case (p_H= 0.05) where the gain was of two nodes.

In order to illustrate the method behavior, circuit *apex6* is considered with periodic variability target and the following parameters: A_1= 0.45, A_2=0.8, popsize |*P*|=50, *tol*= 0.001, n_{add}= 20, p_m= 0.3, p_c=0,4, p_H=0.1. Distribution of the relative error *(F_min-F_best)/F_best* is presented in table 8 and the associated graph in figure 5.

err.	0.1948	0.228916	0.2681	0.2771	0.3393
rel.freq	0.4	0.1	0.2	0.1	0.2
average= 0.2476			std.dev=0.0509	unitized risk=0.2056	

Table 8. Average, standard deviation and unitized risk for (*F_min-F_best*)/*F_best*

The resulted average error is 0.2476. The graph of this distribution is shown in figure 5. The second objective is to get an idea on the stability of the method. The value of the standard deviation obtained for the above-mentioned sample is 0.0509, the unitized risk is 0.2056, and this shows that the method gives good results systematically.

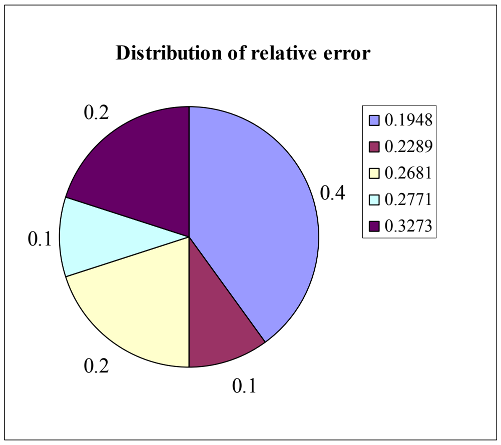


Fig. 5. Distribution of relative error (*F_min-F_best*)/*F_best*

5. FINAL REMARKS

This paper presents a new hybrid GA for optimize variable ordering in OBDDs. It contains new techniques for adapting and for controlling the genetic operators' behavior based on variability target policies. The hybrid GA contains three main improvements: firstly, it uses three mutation operators and three crossover operators applied with given probability distributions. Secondly, a hypermutation operator grafts the sifting technique on the GA. Third, the algorithm contains a control mechanism based on information energy for adapting the population variability. In order to evaluate the GA's performance, experiments were made to determine the adequate algorithm's parameters values, to evaluate the best variability target policy and to study the effect of hypermutation. Experimental results show that the proposed method performs very well [15] and has further development potential.

References

[1] B. Bollig, M. Lobbing, I. Wegener, **Simulated annealing to improve variable orderings for OBDDs**, International Workshop on Logic Synth., pag. 5b:5.1-5.10, 1995.

[2] B. Bollig, I. Wegener, **Improving the Variable Ordering of OBDDs Is NP-Complete**, IEEE Transactions on Computers, vol. 45, 1996.

[3] R.E. Bryant, **Graph-based algorithms for Boolean function manipulation**, IEEE Trans on Computers. 35(8) pag. 667-691, 1986.

[4] K.M.Butler, D. Ross, R. Kapur, M.R. Mercer, **Heuristics to compute variable orderings for efficient manipulation of OBDDs**, Proc. of the 28th Computation Conference, GECCO'02, pag. 942–948. Morgan-Kauffman, 2002.

[5] U.S. Costa, A. M. Moreira, D. Deharbe, **A cache-based parallel genetic algorithm for the bdd variable ordering problem**, Proceedings of SBAC-PAD'2000, pag.99-104, 2000.

[6] U.S. Costa, D. Deharbe, A. M. Moreira, **Variable ordering of bdds with parallel genetic algorithms**, Proceedings of PDPTA'2000, 2000.

[7] R. Drechsler, B. Becker, N. Gockel, **A Genetic Algorithm for Variable Ordering of OBDDs**, IEEE Proceedings, 143(6), pag. 363–368, 1996.

[8] R. Drechsler, N. Gockel, **Minimization of BDDs by Evolutionary Algorithms**, International Workshop on Logic Synthesis, 1997.

[9] S. Droste, D. Heutelbeck, I. Wegener, **Distributed Hybrid Genetic programming for Learning Boolean Functions**, Parallel Problem Solving from Nature – 6th International Conference, pag. 181-190, 2000.

[10] R. Ebendt, F. Gorschwin, R. Drechsler, **Advanced BDD minimization**, Springer, 2005.

[11] R. Ebendt, W.Günther, R.Drechsler, **Combining Ordered Best-First Search with Branch and Bound for Exact BDD Minimization**, 2004, url: informatik.uni-bremen.de.

[12] I. Furdu, **An Analysis of Heuristics for OBDD's Optimization**, VIII ETAI Conference, I1-5, 2007.

[13] S. J. Friedman, K. J. Supowit, **Finding the Optimal Variable Ordering for Binary Decision Diagrams**, IEEE Transactions on Computers, vol. 39, 1990.

[14] D.E Goldberg, **Genetic Algorithms in Search, Optimization and Machine Learning**, Addison Wesley, 1989.

[15] J. Harlow, F. Brglez, **Design of experiments and evaluation in BDD ordering heuristics**, International Journal STTT, (3) pag.193-206, 2001.

[16] W. Lenders, C. Baier, **Genetic Algorithms for Variable Ordering Problem of Binary Decision Diagrams**, url: inf.tu-dresden.de/content/

[17] W. Lenders, **Genetic Algorithms for the Variable Ordering Problem of Binary Decision Diagrams**, Diploma Thesis, Institut für Informatik, Universität Bonn, Germany, 2004.

[18] C. Meinel, T. Theobald, **Algorithms and Data Structures in VLSI Design**, Springer, Berlin, 1998.

[19] C. Meinel, A. Slobodova, **Speeding up variable reordering for OBDDs**, citeseer.ist.psu.edu/cache/papers/cs/640.

[20] O. Onicescu, **Elements of Informational Statistics with Applications**, Technical Editing House, Bucharest, 1979 (in Romanian).

[21] S. Panda, F. Somezi, **Who are the variables in your neighborhood**, International Conference of CAD, pag. 74-77, 1995.

[22] R. Rudell, **Dynamic variable ordering for ordered binary decision diagrams**, International Conference of CAD, pag. 42-47, 1993.

[23] M. A. Thornton, J.P. Williams, R. Drechsler, N. Drechsler, D.M. Wesels, **SBDD Variable Reordering based on Probabilistic and Evolutionary Algorithms**, IEEE Proceedings, Pacific Rim Conference, pag. 381–387, 1999.

[24] CUDD package url: vlsi.colorado.edu/~fabio/CUDD/

[25] LGSynth91 benchmarks at url: http://cadlab.cs.ucla.edu/~kirill/ or cbl.ncsu.edu:16080/benchmarks/

Iulian Furdu

“Vasile Alecsandri” University of Bacău,
Faculty of Sciences
Department of Mathematics and Informatics,
Spiru Haret 8, 600114 Bacău, ROMANIA
e-mail: ifurdu@ub.ro

Octav Brudaru

Institute of Computer Science, Romanian Academy, Iaşi Subsidiary,
“Gh. Asachi” Technical University Iaşi,
Department of Management and Production Systems Engineering, Iaşi, Romania
e-mail: brudaru@tuiasi.ro