An Evolutionary Algorithm for Optimization of Pseudo Kronecker Expressions

Alexander Finder Rolf Drechsler
Institute of Computer Science
University of Bremen
28359 Bremen, Germany
Email: {final,drechsle}@informatik.uni-bremen.de

Abstract—**Using EXOR gates in logic synthesis often results in smaller circuit realizations. While in AND/OR synthesis the problem definition is clear, in AND/EXOR synthesis several classes of optimization problems have been considered. In this context Pseudo Kronecker Expressions (PSDKROs) are highly relevant, since they allow very compact representations while the optimization can be carried out efficiently.**

But the size of PSDKROs depends on a chosen order in which the variables are considered. In this paper an Evolutionary Algorithm (EA) is presented for determining a good decomposition order for PSDKROs. Experimental results are given to demonstrate the efficiency of the approach.

Keywords: ***logic synthesis, AND/EXOR, PSDKRO, evolutionary algorithm, 2-level minimization, BDD, ordering problem***

I. Introduction

The use of EXOR gates in the synthesis process reduces the hardware costs in many cases [25], [24]. Additionally EXOR based circuits often have nice testability properties [18], [20], [21], [5]. Recently, synthesis approaches for quantum circuits have been proposed that also make use of AND/EXOR forms [10], i.e. the expression can be directly mapped to a target circuit.

In contrast to AND/OR minimization — that in the meantime is well understood — in AND/EXOR minimization several restricted classes are considered, like e.g. *Fixed Polarity Reed-Muller Expression* (FPRM) [19] and *Kronecker Expression* (KRO) [6]. (For an excellent overview see [22].) These subclasses are of interest, since the minimization of general *Exclusive Sum of Product Expressions* (ESOPs) turned out to be computationally very hard, i.e. all programs presented so far have long run times and often fail to determine the optimal result (see e.g. [23], [14]).

As one alternative *Pseudo Kronecker Expressions* (PSDKROs) [6], [22] have been proposed, since they are an interesting compromise: the resulting 2-level forms are of moderate size, i.e. close to ESOPs, and additionally the minimization process can be handled within reasonable time bounds. PSDKROs for example are used as a preprocessing step for ESOP minimization to determine a good starting point in [16]. For symmetric functions the optimal PSDKRO can even be determined in polynomial time, as has been shown in [8]. In this paper also an efficient implementation based on *Binary Decision Diagrams* (BDDs) [4] has been proposed.

But, as has already been observed by an example in [22], the size of a PSDKRO is only optimal with respect to a chosen ordering of the variables that is used to decompose the original function. This ordering has not been further studied so far.

Especially for ordering problems EAs have shown to give very good results and a large set of operators, like e.g. PMX [12], is available.

In this paper an EA is presented for determining a good variable ordering for PSDKROs. While only a single example was given in [22] to show that the size of PSDKROs varies dependent on the ordering, here this effect is studied in more detail. The EA is applied for some larger single and multiple output benchmark functions and compared by experiments to the initial order as given in the benchmark. Further experiments are given to show the efficiency of the presented algorithm.

The paper is structured as follows: In Section II PSDKROs are defined and in Section III the algorithm for minimizing PSDKROs for a fixed variable ordering is given. This algorithm from [8] has been extended and is later used for determining the fitness in the EA that is presented in Section IV. In Section V experimental results are given. The paper is finished with a resume of the results in Section VI.

II. Pseudo Kronecker Expressions

In this section briefly the essential definitions of *Pseudo Kronecker Expressions* (PSDKROs) are reviewed and an example for their creation is given. (For more details see [6], [22], [8].)

Let f_i^0 (f_i^1) denote the *cofactor* of function f with respect to $x_i = 0$ ($x_i = 1$) and f_i^2 is defined as $f_i^2 := f_i^0 \oplus f_i^1$, $\oplus$ being the Exclusive OR operation. A Boolean function $f : \mathbb{B}^n \to \mathbb{B}$ then can be represented by one of the following formulae:

$$f = \overline{x}_i f_i^0 \oplus x_i f_i^1 \qquad Shannon\ (S) \tag{1}$$

$$f = f_i^0 \oplus x_i f_i^2 \qquad positive\ Davio\ (pD) \tag{2}$$

$$f = f_i^1 \oplus \overline{x}_i f_i^2 \qquad negative\ Davio\ (nD) \tag{3}$$

If to a function f either S, pD or nD is applied two subfunctions are obtained. To each subfunction again S, pD or nD can be applied. This is done until constant functions are reached. If the resulting expression is multiplied out, a 2-level AND/EXOR form is maintained, called a PSDKRO.

The decompositions are applied with respect to a fixed variable ordering. Notice that the choice of the variable ordering in which the decompositions are applied and the choice of the decomposition per subfunction largely influence the size of the resulting representation [22].

Example 1: Let $f(x_1, x_2, x_3) = x_1 + x_2x_3$. If first f is decomposed using S we get:

$$f^0_{x_1} = x_2x_3 \text{ and } f^1_{x_1} = 1$$

Then $f^0_{x_1}$ is decomposed using nD:

$$(f^0_{x_1})^1_{x_2} = x_3 \text{ and } (f^0_{x_1})^2_{x_2} = x_3$$

Finally, pD is applied for both $(f^0_{x_1})^1_{x_2}$ and $(f^0_{x_1})^2_{x_2}$:

$$((f^0_{x_1})^1_{x_2})^0_{x_3} = 0 \text{ and } ((f^0_{x_1})^1_{x_2})^2_{x_3} = 1$$

$$((f^0_{x_1})^2_{x_2})^0_{x_3} = 0 \text{ and } ((f^0_{x_1})^2_{x_2})^2_{x_3} = 1$$

If we multiply out the expression we obtain:

$$\begin{aligned} f &= (x_3 \oplus \overline{x}_2x_3)\overline{x}_1 \oplus x_1 \\ &= x_1 \oplus \overline{x}_1\overline{x}_2x_3 \oplus \overline{x}_1x_3 \end{aligned}$$

III. Efficient Computation of PSDKROs based on BDDs

For the computation of the optimal PSDKRO for a fixed variable order the algorithm from [8] based on BDDs is used for *reduced ordered BDDs* (ROBDDs). The starting point of the algorithm is the ROBDD representation of the function that has to be minimized. In the following the short term BDD instead of ROBDD is used. Starting from the root of the BDD the graph is recursively traversed and at each node an EXOR operation is carried out. An EXOR operation on BDDs can be performed in polynomial time [4], [2]. Using the fact that for each of the decomposition formulae above two out of the three possible successors f^0_i, f^1_i and f^2_i are needed the two minimizing the resulting PSDKRO are chosen[1]. For each node v a minimal number of terms needed for the representation as a PSDKRO is stored in the variable $v.val$. Thus, each node has to be evaluated only once.

In the following two improvements are presented to speed up the algorithm from [8]. Since this algorithm is the core function for the evaluation of the fitness value in the following, this has a significant influence on the overall run time as will be shown by experiments later (see Section V).

A sketch of the algorithm is given in Fig. 1.

[1]The idea of the algorithm is the same as used in [22] for PSDKRO minimization using *Ternary Decision Diagrams*.

```
psdkro(node $v$, upperbound){
    if($v$ == ZERO) return 0;
    if($v$ == ONE) return 1;
    if($v.val$ defined) return $v$.val;
    node $v_0$ = cofactor$_0$($v$);
    node $v_1$ = cofactor$_1$($v$);
    if($v$ complemented){
        $v_0$ = $\overline{v}_0$;
        $v_1$ = $\overline{v}_1$;
    }
    $c_0$ = psdkro($v_0$);
    $c_1$ = psdkro($v_1$);
    if(prune())
        return $\infty$;
    updateUpperbound();
    node $v_2$ = EXOR($v_0$, $v_1$);
    $c_2$ = psdkro($v_2$);
    $v$.val = $c_0$ + $c_1$ + $c_2$ − max($c_0$, $c_1$, $c_2$);
    return $v$.val;
}
```

Fig. 1. Sketch of the algorithm

A. Complemented Edges

The idea of complemented edges is that a node represents a Boolean function f and its complement $\overline{f}$ coincidently [2]. A complemented edge is an ordinary edge within the decision diagram additionally marked with a complement bit. Since the computation of the complement of a Boolean function takes constant time, the influence of complemented edges on the runtime of the application is minimal.

The use of complemented edges has several advantages [1]. First of all the number of nodes needed to represent a Boolean function ideally can be reduced by half. This implies that there are also less computations necessary for obtaining minimal PSDKROs for subfunctions within the decision diagram.

Each node in the BDD either has two children (low and $high$) or is a leaf with a constant value. Thereby the low child can be complemented. The principal of reducing BDDs by complemented edges is shown in Fig. 2.

If the PSDKRO for the subtree representing $f_{high(v)}$ ($f_{low(v)}$) has been computed a further computation for the subtree of $f_{low(v)}$ ($f_{high(v)}$) is not necessary. Moreover, the result of $f_{high(v)} \oplus f_{low(v)}$ can be directly derived in constant time in case $f_{high(v)} = \overline{f}_{low(v)}$ in using complemented edges, see Fig. 3.

B. Upper Bound

In addition to the root node of a BDD the algorithm described above computes an upper bound which at the beginning is set to ∞. For each node the algorithm first

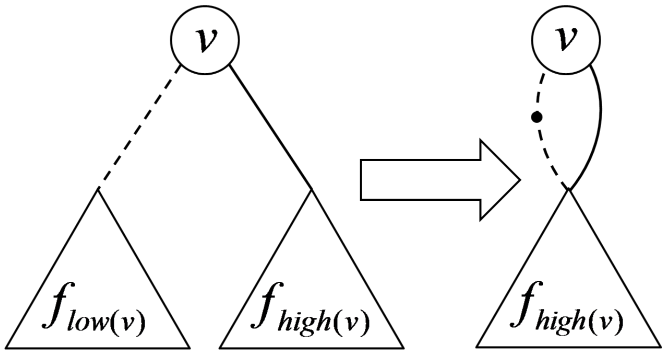

Fig. 2. Principal of reduction by complemented edges.

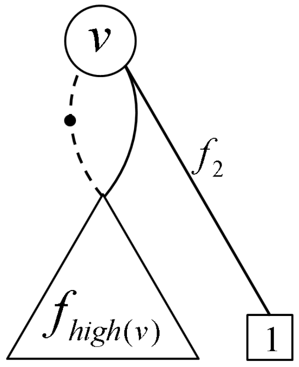

Fig. 3. Fast deriving of f_2 with complemented edges.

computes the minimum PSDKROs for the successors f_i^0 and f_i^1 recursively. If the respective costs c_0 and c_1 exceed the upper bound, the computation is stopped and ∞ is returned because none of the three possible decompositions would create a PSDKRO smaller than the upper bound. Otherwise the upper bound is updated with $max(c_0, c_1)$ if both the cost of v_0 and v_1 are less than the current upper bound. Furthermore the size of the decision diagram that has to be traversed can be reduced by this technique.

IV. Evolutionary Algorithm

In this section a short introduction is given to *Evolutionary Algorithms* (EAs) and the evolutionary operators that are applied to the problem given above are described.

A. Representation

For the implementation integer permutations of length n are used to encode the problem, where n denotes the number of input variables of the considered Boolean function. Each integer string represents a variable ordering. A binary encoding of the problem would require special repair algorithms to avoid the creation of invalid solutions[2]. (The occurring problem is similar to the problem of tour encoding in the *traveling salesman problem*. Similar observations concerning encoding are reported in [27]. The same holds true for algorithms for BDD ordering [9].) As can be seen each integer vector represents a valid ordering. A population is a set of these elements.

[2]In contrast "classical" EAs use a binary encoding of the problem.

B. Objective Function and Selection

As an *objective function* that measures the *fitness* of each element the number of terms in the PSDKRO is used that is computed by the algorithm presented in the previous section.

The parent selection is performed by a deterministic tournament selection. Hereby a tournament between q uniformly chosen individuals is carried out and the best individual is chosen as a parent used for recombination or mutation.

For environment selection (μ+λ)-selection [28] is applied, where the best individuals of the current population and the offspring are chosen equally. This strategy guarantees that the best individual never gets lost and a fast convergency of the EA is obtained. Further parent and environment selection procedures have been tested, like (μ,λ) and stochastic tournament selection, but EA experiments have shown that the methods used are usually advantageous.

C. Initialization

At the beginning of each EA-run an initial population is randomly generated as follows: The first individual represents an integer permutation containing a variable order as it is read from the benchmark file. A copy of this individual is made and for each variable on position $i = 1$ to $n-1$ an exchange with a randomly chosen variable on position r ($r \in [i, n]$) is applied if $i \neq r$. The method guarantees that only valid solutions are generated and additionally it performs very fast.

After that to each element of the initial population a fitness is assigned, i.e. the number of terms of the corresponding PSDKRO.

The following example shows how the generation of random permutations work:

Example 2: An element of length 6 is considered.

Initial individual:	*1*	*2*	*3*	*4*	*5*	*6*

Next, a copy of the initial individual is made and for $i = 1$ to 5 r takes the random numbers (3, 4, 5, 4, 6). The variables on position i and r_i are exchanged, i.e. variable 1 is exchanged with 3, 2 with 4, again 1 with 5 and so on. The resulting permutation determines a newly created element of the initial population.

Step 1, $r = 3$:	$\underline{1}$	2	$\underline{3}$	4	5	6
Step 2, $r = 4$:	3	$\underline{2}$	1	$\underline{4}$	5	6
Step 3, $r = 5$:	3	4	$\underline{1}$	2	$\underline{5}$	6
Step 4, $r = 4$:	3	4	5	$\underline{2}$	1	6
Step 5, $r = 6$:	3	4	5	2	$\underline{1}$	$\underline{6}$
New individual:	3	4	5	2	6	1

D. Evolutionary Operators

In this subsection the evolutionary operators are introduced that are used in the EA. Two crossover operators for the recombination of two parent elements and three mutation operators that change some positions in a single element at random are integrated into the application. These strategies are explained in the following.

1) Crossover: In the presented application *Partially Matched Crossover* (PMX) [12] and *Edge Recombination Crossover* (ERX) [27] are used equally. Both recombination methods create two children from two parents. The parents are selected by the tournament selection described above.

The PMX operator chooses two cut positions at random. Notice that a simple exchange of the parts between the cut positions (as often applied to binary coded EA-problems) is not possible, since this would usually produce invalid solutions, i.e. the resulting element needs not to be a permutation any longer. The operator works as follows to *validate* the children after the exchange:

PMX:
: Construct the children by choosing the part between the cut positions from one parent and preserve the position and order of as many variables as possible from the second parent.

Example 3: For the PMX operator an example is given to explain how the validation process *works. Let* p_1 *and* p_2 *be the parents and let* $i_1 = 3$ *and* $i_2 = 5$ *be the two cut positions. The temporarily constructed child* c_1' *(*c_2'*) is created by copying the part between the cut positions – 4 5 (2 6) – from* p_1 *(*p_2*). Then the characters 5 3 1 and 0 4 (2 0 3 and 1 6) are taken from* p_2 *(*p_1*). As can easily be seen* c_1' *and* c_2' *are invalid solutions. They are validated by exchanging the characters 2 and 4 and also 5 and 6. The resulting children are* c_1 *and* c_2.

PMX:	p_1 :	*2 0 3*	*4 5*	*1 6*
	p_2 :	*5 3 1*	*2 6*	*0 4*
	c_1' :	*5 3 1*	*4 5*	*0 4*
	c_2' :	*2 0 3*	*2 6*	*1 6*
	c_1 :	*6 3 1*	*4 5*	*0 2*
	c_2 :	*4 0 3*	*2 6*	*1 5*

For more details about PMX see [17], [11], [15].

The ERX operator on the contrary does not consider the ordering of the variables but the connections among them. The operator is based on an adjacency matrix, which lists the neighbors of each variable in both parents. Beginning with a randomly chosen variable of a parent element next a variable is chosen at random which has the smallest neighbor set. Thereby already chosen variables are removed from all the neighbor sets. This procedure is repeated until all variables are appended to the newly created individual.

Also further crossover operators have been tested, like *merging* [17], *ordered* [15] and *cycle crossover* [7], but they did not improve the results obtained.

2) Mutation: Three different *mutation operators* are used as follows:

Mutation (SWAP):
: Select two positions of a parent at random and perform the exchange of the values of these two positions.

Mutation with neighbor (NEIGH):
: Choose one position i randomly. Then perform SWAP at positions i and $i + 1$.

Mutation with inversion (INV):
: Select two positions $i < j$ at random. The difference $j - i$ should be at the maximum not larger as the third size of an individual. Invert all variables within i and j.

Example 4: For the operators SWAP and INV examples are given to show the strategy of the mutation operators. Let 2 *and* 5 *in both cases be the chosen positions. The child results from exchanging the corresponding values for SWAP respectively inverting the variables between positions 2 and 5 for INV.*

SWAP:	p :	$6\ \underline{3}\ 1\ 2\ \underline{5}\ 0\ 4\ 9\ 7\ 8$
	c :	*6 5 1 2 3 0 4 9 7 8*
INV:	p :	$6\ \underline{3\ 1\ 2\ 5}\ 0\ 4\ 9\ 7\ 8$
	c :	*6 5 2 1 3 0 4 9 7 8*

The mutation operators are a generalization of exchanging neighboring variables that is the basic operation for dynamic variable ordering.

E. Algorithm

Using the operators introduced above the EA works as follows:

- Initially a random population of finite integer strings is generated. Each of these strings corresponds to a variable ordering of the BDD.
- An offspring of the same size of the parent population is created in each iteration. This is done by applying the genetic operators described above. Then the newly created elements are evaluated.
- According to the fitness the best individuals of both populations are chosen to build the next generation. After each iteration the size of the population is constant (*steady-state reproduction*).
- The optimization process stops in three cases. The termination criterions are chosen based on experiments in a way that the EA provides a compromise between acceptable runtime and high quality results:

```
evolutionary_algorithm(benchmark){
    generate_random_population();
    calculate_fitness();
    {
        select_parents();
        recombine_and_mutate();
        calculate_fitness();
        $\mu+\lambda$_selection();
    }while(stopping criterion unfilled)
    return best_individual();
}
```

Fig. 4. Sketch of the basic evolutionary algorithm

1) No improvement is obtained for

$$20 \cdot \textit{ln(number of variables)}$$

iterations.
2) A maximum number of 500 iterations has been carried out. The number of generations also can be parameterized by the user.
3) The best individual has reached a given lower bound.

A sketch of the evolutionary algorithm is given in Figure 4.

F. Parameter Settings

The size of the population is chosen two times larger than the number of variables of the considered Boolean function, if the number of input variables is smaller or equal to 25. For larger functions the population size is set constant to 50, since otherwise the EA is too time consuming.

PMX and ERX are applied with a probability of 35% while the different mutation operators are used with a probability of 65% to create the offspring of a population. The probability values have been determined by experiments and are a good compromise between runtime and the quality of results. Mutation also could be carried out on newly elements that are created from PMX or ERX. However, the probability of all types of operators and the size of the population also can be parameterized by the user.

V. Experimental Results

In this section experimental results are presented based on several benchmark functions out of the LGSynth91 benchmark set [29]. The techniques described above have been implemented in C/C++ using the BDD package CUDD [26]. The evolutionary operators and algorithm have been embedded into the EO library [13]. All runtime measures are given in CPU seconds on an *Intel Core 2 Duo E6700* workstation with 4096 MByte of main memory.

In Table I the results of a maximizing and a minimizing EA are compared to show the influence of the variable ordering on the size of the corresponding PSDKROs. The results for the maximizing algorithm are obtained by changing the minimizing fitness function to a maximizing one.

TABLE I
Minimizing and maximizing EA.

Benchmark			EA_{min}	EA_{max}
name	in	out		
co14	14	1	14	14
sym9	9	1	90	90
sym10	10	1	134	134
5xp1	7	10	48	60
alu4	14	8	498	917
clip	9	5	100	146
duke2	22	29	194	265
misex3	14	14	851	1444
sao2	10	4	50	64
vg2	25	8	184	326

As can be seen for the first three benchmarks in the table both algorithms carry out identical results. The reason for this is that the concerned benchmarks are representing totally symmetric Boolean functions. This implies that the variable ordering has no effect on the size of the corresponding PSDKROs [8].

Concerning the remaining asymmetrically functions of the table it is obvious that the results delivered from maximizing EA (EA_{max}) in almost all cases have twice the size as the results obtained from minimizing EA (EA_{min}). Especially for the benchmarks `alu4` and `misex3` the difference between the minimal and the maximal EA solution is remarkable.

In Table II the results delivered from the EA are compared with PSDKROs created by using the initial order (IO) as given in the benchmark. This is done to demonstrate the advance of the EA towards the standalone algorithm like it is used in [8], [16], [24].

Further the AND/EXOR representations are compared with *Sum of Product Expressions* (SOPs) generated by *ESPRESSO* [3]. It can be seen that the EA often generates much better PSDKROs with respect of their size than the conventional algorithm. It is also striking that the AND/EXOR representations in many cases contain fewer cubes than the corresponding AND/OR expressions. Particularly for *t481* and *add6* there exists much smaller PSDKROs than SOPs.

Next, in Table III the influence of the use of the upper bound (see Section III-B) is investigated. Therefore the fitness computation in the EA first has been performed using an upper bound (pruning) and after that without. The number of cuts in the table depicts the overall number of aborts during the evaluation process of an EA run, i.e. the stopping of recursive traversals through the BDD during fitness computation. It can be seen clearly that for larger benchmark instances often several 100000 cuts are carried out within a single run of the EA. The average number of cuts diverges in dependence on the output function and the variable ordering. It has also been

TABLE II
NUMBER OF TERMS.

Benchmark			SOP	IO		EA	
name	in	out		cubes	time	cubes	time
co14	14	1	14	14	0.01	14	0.03
rd84	8	4	255	90	0.01	90	0.05
sym9	9	1	84	90	0.01	90	0.02
sym10	10	1	210	134	0.01	134	0.04
5xp1	7	10	65	48	0.01	48	0.17
add6	12	7	355	132	0.02	132	0.38
alu4	14	8	575	677	0.06	498	5.26
clip	9	5	120	113	0.01	100	0.10
duke2	22	29	86	198	0.11	194	7.53
misex3	14	14	690	1056	0.49	851	8.63
sao2	10	4	58	54	0.01	50	0.09
t481	16	1	481	13	0.01	13	0.01
vg2	25	8	110	324	0.09	184	4.80

TABLE III
COMPARISON OF EA WITH AND WITHOUT PRUNING.

Benchmarks	Pruning EA		EA
Name	#cuts	time	time
add6	> 1200	0.38	0.54
alu4	> 300000	5.26	5.57
duke2	> 380000	7.53	7.89
misex3	> 520000	8.63	9.65
sym10	> 800	0.04	0.04
vg2	> 340000	4.8	5.29

determined that for some functions only few cuts are carried out while for others hundreds of cuts per variable ordering are performed. In almost all cases the runtime of the pruning EA is better than the runtime of the EA without pruning and many cuts are performed. Thus the advance of the algorithm by using an upper bound is distinctive.

VI. CONCLUSIONS

In this paper, an evolutionary algorithm for the optimization of PSDKROs has been presented. A BDD based implementation was given. The results of the EA have been compared to the conventional algorithm which does not consider an optimization of the variable ordering. Further the speed-up techniques introduced in the section for the computation of PSDKROs based on BDDs have been evaluated.

The experimental results achieved show that the proposed EA is a practical approach to create small PSDKROs which further may be used for computing a good starting cover for the minimization of general ESOPs. The presented algorithm is also applicable for functions with more than 20 variables.

REFERENCES

[1] S.B. Akers. Binary decision diagrams. *IEEE Trans. on Comp.*, c-27(6):509–516, June 1978.

[2] K.S. Brace, R.L. Rudell, and R.E. Bryant. Efficient implementation of a BDD package. In *Design Automation Conf.*, pages 40–45, 1990.

[3] R.K. Brayton, G.D. Hachtel, C. McMullen, and A.L. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Publishers, 1984.

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

[5] M. Chatterjee, D. K. Pradhan, and W. Kunz. LOT: logic optimization with testability - new transformations using recursive learning. In *Int'l Conf. on CAD*, pages 318–325, 1995.

[6] M. Davio, J.P. Deschamps, and A. Thayse. *Discrete and Switching Functions*. McGraw-Hill, 1978.

[7] L. Davis. *Handbook of Genetic Algorithms*. van Nostrand Reinhold, New York, 1991.

[8] R. Drechsler. Pseudo kronecker expressions for symmetric functions. In *VLSI Design, 1997. Proceedings., Tenth International Conference on*, pages 511–513, Jan 1997.

[9] R. Ebendt, G. Fey, and R. Drechsler. *Advanced BDD Optimization*. Springer, 2005.

[10] K. Fazel, M.A. Thornton, and J.E. Rice. Esop-based toffoli gate cascade generation. In *Communications, Computers and Signal Processing, 2007. PacRim 2007. IEEE Pacific Rim Conference on*, pages 206–209, Aug. 2007.

[11] D.E. Goldberg. *Genetic Algorithms in Search, Optimization & Machine Learning*. Addision-Wesley Publisher Company, Inc., 1989.

[12] D.E. Goldberg and R. Lingle. Alleles, loci, and the traveling salesman problem. In *Proceedings of an International Conference on Genetic Algorithms and their Applications*, pages 154–159, 1985.

[13] M. Keijzer, J. J. Merelo, G. Romero, and M. Schoenauer. Evolving Objects: a general purpose evolutionary computation library. In *5th International Conference on Artificial Evolution*, 2001.

[14] T. Kozlowski, E. L. Dagless, and J. M. Saul. An enhanced algorithm for the minimization of exclusive-or sum-of-products for incompletely specified functions. In *Int'l Conf. on Comp. Design*, pages 244–249, 1995.

[15] Z. Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs*. Springer-Verlag, 1994.

[16] Alan Mishchenko and Marek Perkowski. Fast Heuristic Minimization of Exclusive-Sums-of-Products. In *Proc. Reed-Muller Workshop '01*, pages 242–250, 2001.

[17] I.M. Oliver, D.J. Smith, and J.R.C. Holland. A study of permutation crossover operators on the traveling salesman problem. In *International Conference on Genetic Algorithms*, pages 224–230, 1987.

[18] S.M. Reddy. Easily Testable Realizations for Logic Functions. *IEEE Trans. on Comp.*, c-21(11):1183–1188, Nov. 1972.

[19] I.S. Reed. A class of multiple-error-correcting codes and their decoding scheme. *IRE Trans. on Inf. Theory*, 3:38–49, 1954.

[20] K.K. Saluja and S.M. Reddy. Fault Detecting Test Sets for Reed-Muller Canonic Networks. *IEEE Trans. on Comp.*, c-24:995–998, 1975.

[21] A. Sarabi and M.A. Perkowski. Design for testability properties of AND/XOR networks. *IFIP WG 10.5 Workshop on Applications of the Reed-Muller Expansion in Circuit Design*, pages 147–153, 1993.

[22] T. Sasao. AND-EXOR expressions and their optimization. In T. Sasao, editor, *Logic Synthesis and Optimization*, pages 287–312. Kluwer Academic Publisher, 1993.

[23] T. Sasao. EXMIN2: A Simplification Algorithm for Exclusive-OR-Sum-of Products Expressions for Multiple-Valued-Input Two-Valued-Output Functions. *IEEE Trans. on CAD*, 12(5):621–632, May 1993.

[24] T. Sasao. *Logic Synthesis and Optimization*. Kluwer Academic Publisher, 1993.

[25] J. Saul, B. Eschermann, and J. Frössl. Two-level logic circuits using EXOR sums of products. *IEE Proceedings*, 140:348–356, 1993.

[26] F. Somenzi. Efficient manipulation of decision diagrams. *STTT*, 3(2):171–181, 2001.

[27] D. Whitley, T. Starkweather, and D. Fuquay. Scheduling problems and traveling salesman: The genetic edge recombination operator. In *International Conference on Genetic Algorithms*, pages 133–140, 1989.

[28] Darrell Whitley. The genitor algorithm and selection pressure: Why rank-based allocation of reproductive trials is best. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 116–121. Morgan Kaufmann, 1989.

[29] S. Yang. Logic synthesis and optimization benchmarks user guide. Technical Report 1/95, Microelectronic Center of North Carolina, 1991.