18

Evolutionary Algorithms in Decomposition-Based Logic Synthesis

Mariusz Rawski
Warsaw University of Technology
Poland

1. Introduction

Functional decomposition is a logic synthesis method that has recently gained much recognition. The main reason is the evolution of field programmable gate-arrays (FPGAs) as a new technology for digital system implementation. Architecture of FPGA is based on the lookup table (LUT) as basic building block. An *n*-input LUT is capable of implementing any Boolean function of up to *n* variables. Thus, logic synthesis for LUT-based FPGAs must transform a logic network into network that consists of nodes with up to *n* inputs only. Each node of such network can be then implemented by a single LUT. For this reason, for the case of implementation targeting FPGA structure, decomposition is a very efficient method. Modern FPGA devices have very complex structure. Today's FPGAs are entire programmable systems on a chip (SoC) which are able to cover an extremely wide range of applications. The Altera Stratix III and Xilinx Virtex-5 families of devices, both using a 65 nm manufacture process, can be used as examples of contemporary FPGAs. The basic architecture of FPGAs has not changed dramatically since their introduction in the 1980s. Early FPGAs used a logic cell consisting of a 4-input lookup table and register. Present devices employ larger numbers of inputs (6-input for Virtex-5 and 7-input for Stratix III) and have other associated circuitry. Another enhancement extensively used in modern FPGAs are specialized embedded blocks, serving to improve delay, power and area if utilized by the application, but waste area and power if unused. Early embedded blocks included fast carry chains, memories, phase locked loops, delay locked loops, boundary scan testing and multipliers. More recently, multipliers have been replaced by digital signal processing (DSP) blocks which add support for logical operations, shifting, addition, multiply-add, complex multiplication etc. Some architectures even contain hardware CPU cores. This greatly extends the space of possible solution during the process of mapping the design into FPGA structure with such embedded blocks. Unfortunately such heterogeneous structure of available logic resources greatly increases the complexity of mapping algorithms. The existing CAD tools are not well suited to utilize all possibilities that such modern programmable structures offer due to the lack of appropriate logic synthesis methods.
Functional decomposition is perceived as one of the best logic synthesis methods targeted FPGAs. It relies on breaking down a complex system into a network of smaller and relatively independent co-operating subsystems, in such a way that the original system's behavior is preserved. A system is decomposed into a set of smaller subsystems, such that each of them is easier to analyze, understand and synthesize. Decomposition allows

synthesizing the Boolean function into multilevel structure that is built of components, each of which is in the form of LUT logic block specified by truth tables.

Since the Ashenhurst-Curtis decomposition have been proposed, the research has been focused in forming new decomposition techniques (Łuba & Selvaraj, 1995; Sasao et al., 2001; Scholl, 2001; Brzozowski & Łuba, 2003; Rawski, 2007a). The researchers have developed many types of decompositions, but they are still based on Ashenhurst's ideas. Thanks to the fact that the functional decomposition gives very good results in the logic synthesis of combinational circuits, it is viewed for the most part, as a synthesis method for implementing combinational functions into FPGA-based architectures (Wurth et al, 1999; Scholl, 2001; Rawski et al., 2007). However, the decomposition-based method can be used beyond this field. Decomposition-like synthesis methods are not limited only to logic synthesis of digital circuits. The strong motivation for developing decomposition techniques comes recently from modern research areas such as pattern recognition, knowledge discovery and machine learning in artificial intelligence (Perkowski et al. 1997).

The practical usefulness of functional decomposition for very complex systems is limited by the lack of an efficient method for the construction of the high quality subsystems. In the subsystem construction process the following three factors play an extremely important role: an appropriate input support selection for subsystems, decision which (multi-valued) function will be computed by a certain subsystem and encoding of the subsystem's function with binary output variables. For large functions the solution space is so huge that heuristic method for solving this problem has to be used. This is an NP-hard problem and thus heuristic methods have to be used to efficiently and effectively search for optimal or near-optimal solutions.

There are two types of algorithms solving input variable partitioning problem. The algorithms finding decompositions without using any search heuristics. The basic idea of these algorithms is to limit the search to some input variable partitions. This is done by using different functional methods to choose which partitions will be evaluated. These methods select partitions through Reed-Muller expansions, Fourier transforms, binary difference equations, and technology-based mappings (Łuba et al., 1995; Perkowski, 1994; Steinbach & Stokert, 1994). The second type of algorithms utilize different heuristic methods. In (Rawski et al., 2001) input variable partitioning method based on information relationship measures was presented, which produced optimal or sub-optimal results for factions of considerable size.

In recent years the use of the genetic algorithms has received widespread attention. An evolutionary computing is inspired by Darwin's theory of evolution. In other words, problems are solved by an evolutionary process resulting in the best (fittest) solution (survivor) – the solution is evolved. 'Genetic algorithm' term was introduced by John Holland (Holland, 1975). The evolutionary algorithm is one of heuristics, which not necessarily provides the best possible solution. However, these sub-optimal solutions are considered as acceptable, because in many problems it is not possible to find the best solution in reasonable time. It means that evolutionary algorithms are especially useful for problems with a vast search space and non-polynomial time algorithms solving the given problem.

The evolutionary algorithms need individuals that represent a solution attempt to the problem they are trying to solve. The population needs to be tested to find how well individuals perform, and new individuals are created that are combinations of existing good solutions with some occasional variations. The cycle of testing and creation of new

individuals is repeated until a suitable solution is found, all the individuals represent the same solutions, or the search is abandoned.
This approach has been used to find approximate solutions to NP-complete optimization problems (Khuri, 1994). There have been attempts to apply genetic algorithms to functional decomposition (Noviskey et al., 1994). In (Rawski et al., 2004) the application of evolutionary algorithms was proposed to solve input support selection problem for functional decomposition based on blanket calculus. The solution has been extended to decomposition based on BDDs (Morawiecki & Rawski, 2008)
In this chapter an application of evolutionary algorithm for functional decomposition-based logic synthesis will be discussed. First an introduction to functional decomposition based on cubes and BDDs will be given. Next basics of evolutionary algorithms will be outlined. Subsequently the heuristic input partitioning method will be presented. Following that some experimental results will be discussed. The experimental results demonstrate that the proposed method is able to construct optimal or near optimal decompositions efficiently, even for large systems.

2. Basic information

In this section, only information that is necessary for an understanding of this chapter is reviewed. More detailed description of functional decomposition based on partition calculus can be found in (Brzozowski & Łuba, 2003), functional decomposition based on BDD in (Scholl, 2001).

2.1 Functional decomposition

The set X of input variables of Boolean function is partitioned into two subsets: *free variables* U and *bound variables* V, such that $U \cup V = X$. Assume that the input variables $x_1, ..., x_n$ have been relabeled in such a way, that:
$U = \{x_1, ..., x_r\}$ and
$V = \{x_{n-s+1}, ..., x_n\}$.
Consequently, for an n-tuple x, the first r components are denoted by x^U and the last s components are denoted by x^V.

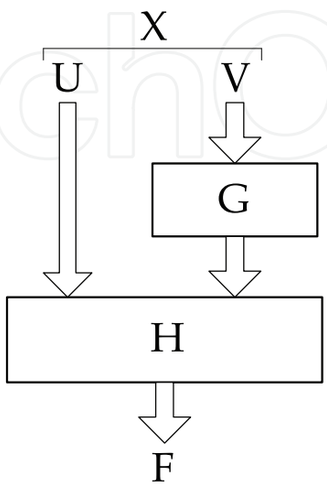


Fig. 1. Schematic representation of the functional decomposition.

Let F be a Boolean function with n inputs and m outputs and let (U, V) be the pair of sets defined above. Assume that F is specified by a set of the function's cubes. Let G be a function with s inputs and p outputs, and let H be a function with $r + p$ inputs and m outputs. The pair (G, H) represents a serial decomposition of F with respect to (U, V), if for every minterm b relevant to F, $G(b^V)$ is defined, $G(b^V) \in \{0, 1\}^p$, and $F(b) = H(b^U, G(b^V))$. G and H are called blocks of the decomposition (Fig. 1).

2.2 Functional decomposition based on blanket calculus

A Boolean function can be specified using the concept of cubes (input patterns) representing some specific sub-sets of minterms (Tab. 1.). In a minterm, each input variable position has a well-specified value. In a cube, positions of some input variables can remain unspecified and they represent "any value" or "don't care" (–). A **cube** may be interpreted as a p-dimensional subspace of the n-dimensional Boolean space or as a product of $n - p$ variables in Boolean algebra (p denotes the number of components that are '–'). For function from Table 1 truth table with $2^4 = 16$ rows would be required to describe the function using minterms. Since cube represents a set of minterms, application of cubes allows for much more compact description in comparison with minterm representation. For example cube 10–0 from row 2 of truth table from Table 1 represents set of two minterms {1000, 1010 }.

	x_1	x_2	x_3	x_4	y
1	0	0	–	0	1
2	1	0	–	0	1
3	–	0	0	–	1
4	–	–	1	1	0
5	–	1	1	0	0
6	1	1	–	1	0
7	0	–	0	1	1
8	–	1	0	0	0

Table 1. Example function.

For pairs of cubes and for a certain input subset B, we define the **compatibility relation** COM as follows: each two cubes S and T are compatible (i.e. $S, T \in \text{COM}(B)$) if and only if $x(S) \sim x(T)$ for every $x \subseteq B$. The compatibility relation ~ on {0, –, 1) is defined as follows [1]: 0 ~ 0, – ~ –, 1 ~ 1, 0 ~ –, 1 ~ –, – ~ 0, – ~ 1, but the pairs (1, 0) and (0, 1) are not related by ~. The compatibility relation on cubes is reflexive and symmetric, but not necessarily transitive. In general, it generates a "partition" with non-disjoint blocks on the set of cubes representing a certain Boolean function F. The cubes contained in a block of the "partition" are all compatible with each other.

"Partitions" with non-disjoint blocks are referred to as blankets (Brzozowski & Łuba, 2003). The concept of blanket is a simple extension of ordinary partition and typical operations on blankets are strictly analogous to those used in the ordinary partition algebra.

Definition 1. Blanket

A **blanket** on a set S is such a collection of (not necessary disjoint) distinct subsets B_i of S, called blocks, that

$$\bigcup_i B_i = S \tag{1}$$

Each block B_i of blanket has its cube representative $r(B_i)$ that indicates the value of variables inducing blanket corresponding to this block.

Example 1 (Blanket-based representation of Boolean functions).

For function F from Table 1, the blankets induced by particular input and output variables and by the two-output function on the set of function F's input patterns (cubes) are as follows:

$$\beta_{x1} = \{\overline{1,\ 3,\ 4,\ 5,\ 7,\ 8};\ \overline{2,\ 3,\ 4,\ 5,\ 6,\ 8}\}, \tag{2}$$

$$\beta_{x2} = \{\overline{1, 2, 3, 4, 7};\ \overline{4, 5, 6, 7, 8}\}, \tag{3}$$

$$\beta_{x3} = \{\overline{1, 2, 3, 6, 7, 8};\ \overline{1, 2, 4, 5, 6}\}, \tag{4}$$

$$\beta_{x4} = \{\overline{1, 2, 3, 5, 8};\ \overline{3, 4, 6, 7}\}, \tag{5}$$

$$\beta_{y} = \{\overline{4, 5, 6, 8}\ ;\ \overline{1, 2, 3, 7}\}. \tag{6}$$

The product of two blankets β_1 and β_2 is defined as follows:

$$\beta_1 \bullet \beta_2 = \{\ B_i \cap B_j \mid B_i \in \beta_1 \text{ and } B_j \in \beta_2\ \}. \tag{7}$$

For two blankets we write $\beta_1 \le \beta_2$ if and only if for each B_i in β_1 there exists a B_j in β_2 such that $B_i \subseteq B_j$. The relation $\le$ is reflexive and transitive.

For example:

$$\beta_{x2x3} = \beta_{x2} \bullet \beta_{x3} = \{\overline{1, 2, 3, 7};\ \overline{1, 2, 4};\ \overline{6, 7, 8};\ \overline{4, 5, 6}\ \}, \tag{8}$$

$$\beta_{x2x3} \le \beta_{x2}. \tag{9}$$

Information on the input patterns of a certain function F is delivered by the function's inputs and used by its outputs with precision to the blocks of the input and output blankets. Knowing the block of a certain blanket, one is able to distinguish the elements of this block from all other elements, but is unable to distinguish between elements of the given block. In this way, information in various points and streams of discrete information systems can be modeled using blankets.

Theorem 1. Existence of the serial decomposition (Brzozowski & Łuba, 2003).

Let F be a Boolean function with n inputs and m outputs and let (U, V) be the pair of sets: *free variables* U and *bound variables* V, such that $U \cup V = X$. Let β_V, β_U, and β_F be blankets induced on the function F's input cubes by the input sub-sets V and U, and outputs of F, respectively.

If there exists a blanket β_G on the set of function F's input cubes such that $\beta_V \le \beta_G$, and $\beta_U \bullet \beta_G \le \beta_F$, then F has a serial decomposition with respect to (U, V).

Proof of Theorem 1 can be found in (Brzozowski & Łuba, 2003).

As follows from Theorem 1 the main task in constructing a serial decomposition of a function F with given sets U and V is to find a blanket β_G which satisfies the condition of the theorem. Since β_G must be $\ge \beta_V$, it is constructed by merging blocks of β_V as much as possible.

Two blocks B_i and B_j of blanket β_V are compatible (mergeable), if blanket γ_{ij} obtained from blanket β_V by merging B_i and B_j into a single block satisfies the second condition of Theorem 1, that is, if $\beta_U \bullet \gamma_{ij} \leq \beta_F$. Otherwise blocks B_i and B_j are incompatible (unmergeable). A subset δ of blocks of the blanket β_V is a compatible class of blocks if the blocks in δ are pair wise compatible. A compatible class is maximal if it is not contained in any other compatible class.

From the computational point of view, finding maximal compatible classes is equivalent to finding maximal cliques in a graph $\Gamma = (N, E)$, where the set N of vertices is the set of blocks of β_V and set E of edges is formed by set of compatible pairs.

The next step in the calculation of β_G is the selection of a set of maximal classes, with minimal cardinality, that covers all the blocks of β_V. The minimal cardinality ensures that the number of blocks of β_G, and hence the number of outputs of the function G, is as small as possible.

In certain heuristic strategies, both procedures (finding maximal compatible classes and then finding the minimal cover) can be reduced to the graph coloring problem.

Calculating β_G corresponds to finding the minimal number k of colors for graph $\Gamma = (N, E)$.

Example 2. For the function from Table 1 specified by a set F of cubes numbered 1 through 8, consider a serial decomposition with $U = \{x_1\}$ and $V = \{x_2, x_3, x_4\}$.

We find

$$\beta_U = \beta_{x1} = \{\overline{1,3,4,5,7,8}; \overline{2,3,4,5,6,8}\}, \tag{10}$$

$$\beta_V = \beta_{x2x3x4} = \{\overline{1,2,3}; \overline{3,7}; \overline{1,2}; \overline{4}; \overline{6,7}; \overline{5}; \overline{4,6}\}, \tag{11}$$

$$\beta_F = \beta_y = \{\overline{4, 5, 6, 8}; \overline{1, 2, 3, 7}\}. \tag{12}$$

Let β_G be as follows:

$$\beta_G = \{\overline{1,2,3,7}; \overline{4,5,6,8}; \overline{6,7}\}. \tag{13}$$

It is easily verified that β_G satisfies the condition of Theorem 1 (more detailed description of partition calculus can be found in (Brzozowski & Łuba, 2003)). Thus function F has a serial decomposition with respect to (U, V).

Number of blocks in blanket β_G determines the number of outputs of block G:

$$p = \lceil log_2(q) \rceil, \tag{14}$$

where q is the number of blocks in blanket β_G.

Since in example β_G has 3 blocks, to encode blocks of this blanket two encoding bits g_1 and g_2 have to be used. To define a function G by a set of cubes we calculate cube representatives, $r(B_i)$, assigned to each block B_i of β_V. The relationship between blocks of β_V and their cube representatives, $r(B_i)$, relies on containment of block B_i in blocks of β_{xj} from $x_j \in V$. Finally, the value of function G is obtained on the basis of containment of blocks B_i in blocks of β_G.

To compute the cubes for function H we consider each block of the product $\beta_U \bullet \beta_G$. Their representatives are calculated in the same fashion. Finally, the outputs of H are calculated with respect to β_F

The process of functional decomposition based on blanket calculus consists of the following steps:

- the selection of an appropriate input support V for block G (input variable partitioning),
- the calculation of the blankets β_U, β_V and β_F,
- the construction of an appropriate multi-block blanket β_G (this corresponds to the construction of the multi-valued function of block G),
- the creation of the binary functions H and G by representing the multi-block blanket β_G as the product of a number of certain two-block blankets (this is equivalent to encoding the multi-valued function of block G defined by blanket β_G with a number of binary output variables).

2.3 Functional decomposition based on BDDs

A Boolean function can be represented using binary decision diagrams. BDDs as a method of representation of single-output Boolean functions were introduced by Lee (Lee, 1959) and later Ackers (Ackers, 1978).

Definition 2. Binary decision diagram (BDD)

Binary decision diagram is a rooted directed acyclic graph $\Gamma = (V, E)$ with node (vertex) set V and arc set E. The graph has terminal nodes called leaves. To each leaf node there is assigned a value 0 or 1. Each non terminal node $v \in V$ is labeled with a Boolean variable $var(v)$ and has arcs directed towards two children: $low(v) \in V$ corresponding to the case where the variable is assigned 0, and $high(v) \in V$ corresponding to the case where the variable is assigned 1.

When a Boolean function is represented by binary decision diagram with a given assignment to the variables, the value yielded by the function is determined by tracing a path from the root to a terminal vertex, following the branches indicated by the values assigned to the variables. The function value is then given by the terminal vertex label.

Definition 3. Ordered binary decision diagram (OBDD)

An ordered binary decision diagram is a BDD where an ordering < over set of variables is defined, and for any node v and either nonterminal child u, their respective variables must be ordered $var(v) < var(u)$.

In (Bryant, 1986) Bryant presented algorithms that efficiently manipulated BDDs assuming ordering of the variables. He developed a method to reduce the size of BDDs by removing 'redundant' nodes and subgraphs which occur more than once. Bryant also proved that the reduced representation is canonical in respect to a given variable ordering.

Definition 4: **Reduced Ordered Binary Decision Diagram (ROBDD)** is an OBDD, that has no vertex v such that $low(v) = high(v)$ and for no pair $\{u, v\}$ sub-graphs rooted in v and u are isomorphic.

Binary decision diagrams made it possible to develop new algorithms for decomposition, feasible for much larger functions than previously possible. In a BDD, the decomposition can be easily computed by moving the bound variables V to the upper part of the graph and counting the number of children below the boundary line, usually called cut line.

Definition 5. Cut-set

Let Γ be the ROBDD representing a function F with variable ordering O, let $cut\_set(\Gamma, O, l)$ denote the set of nodes whose levels are greater than l and have edges from nodes of level lower or equal to l (top node has level 1).

Theorem 2. Existence of the serial decomposition.
Let F be Boolean function and (U, V) be the pair of sets: *free variables* and *bound variables*. Let Γ be ROBDD representing function F with variable ordering such that bound variables are in upper part of Γ. Let

$$p=\lceil \log_2(|\, cut\_set(\Gamma,O,l)\,|)\rceil \tag{15}$$

If $p < l$, there exists decomposition in the form $F(X) = H(U, G(V))$, where function G has p outputs.
The size of *cut_set* from (15) plays the same role in BDD-based function decomposition as number of blocks in blanket β_G from (14).
Detailed description of functional decomposition based on BDD can be found in (Scholl, 2001).
Decomposition algorithms following a BDD-cut strategy proved to be orders of magnitude faster than those based on decomposition charts and cube representations. However, they require a reordering of the BDD to move the target set of variables to the top of the graph.

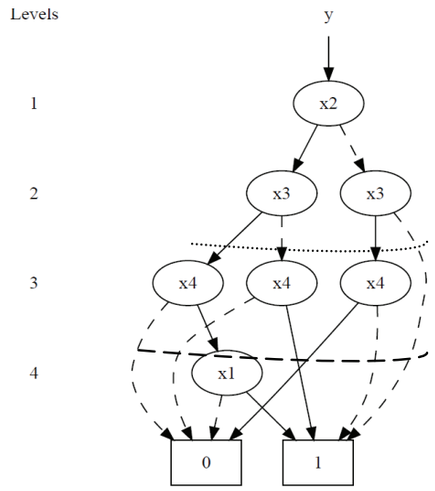


Fig. 2. ROBDD for function from Table 1.

Example 3. The ROBDD diagram Γ presented on Fig. 2 represents function F from Table 1 for ordering O={ x_2, x_3, x_4, x_1}. Let consider two cut-lines: at level 2 (dotted line) and at level 3 (dashed line). We have:

$$q_1=|\, cut\_set(\Gamma,O,2)\,| = 4, \tag{16}$$

$$q_2=|\, cut\_set(\Gamma,O,3)\,| = 3. \tag{17}$$

Following (15):

$$p_1=\lceil \log_2(q_1)\rceil=2, \tag{18}$$

$$P_2=\lceil \log_2(q_2)\rceil=2. \tag{19}$$

According to Theorem 2 decomposition with U = {x_4, x_1} and V = {x_2, x_3} does not exist since p_1 does not satisfy condition $p_1 < l$, where l = 2. Block G would require 2 outputs, while having 2 inputs.
However there exists decomposition with U = {x_1} and V = {x_2, x_3, x_4}, since p_1= 2 and l = 3. The size of block G will be 3 inputs and 2 outputs.

2.4 Evolutionary algorithms

An evolutionary computing is inspired by Darwin's theory of evolution. In other words, problems are solved by an evolutionary process resulting in the best (fittest) solution (survivor) – the solution is evolved. 'Genetic algorithm' term was introduced by John Holland (Holland, 1975). Here an evolutionary algorithm is used, which is more general term.

The evolutionary algorithm is the heuristics, which not necessarily provides the best possible solution. However, these sub-optimal solutions are considered as acceptable, because in many problems it is not possible to find the best solution in reasonable time. It means that evolutionary algorithms are especially useful for problems with a vast search space and non-polynomial time algorithms solving the given problem.

The evolutionary algorithms need individuals that represent a solution attempt to the problem they are trying to solve. The construction of an algorithm starts with mapping a problem into a set of chromosome representations. The population needs to be tested to find how well individuals perform, and new individuals are created that are combinations of existing good solutions with some occasional variations. The cycle of testing and creation of new individuals is repeated until a suitable solution is found, all the individuals represent the same solutions, or the search is abandoned. The basic steps of an evolutionary algorithm are presented on Fig. 3.

To construct the algorithm following qualities have to be defined:

- a population of individuals, where each individual represents an encoded form of a possible solution to the problem being solved,
- methods for testing individual solutions and assigning fitness (how good the solution is),
- methods for selecting suitable parents that will be used to produce new individuals (offspring),
- methods for manipulating the encoded forms of individuals, often called "genetic operators"; these operators are used to create new children from parents (for example, "crossover" techniques), and for introducing other variations (such as "mutation") into the population,
- parameters to manipulate the probability and effect of operators.

```
Evolutionary algorithm()
begin
    t := 0
    P^0 := create_initial_population()
    evaluate_fitnes(P^0)
    while (no_improvement_iterations > threshold) do
    begin
        T^t := selection_operator (P^t)
        O^t := crossover_operator (T^t)
        evaluate_fitnes(O^t)
        if (mutation_condition) then
            O^t := mutation_operator (O^t)
        P^(t+1) := O^t
        t := t +1
    end
end
```

Fig. 3. Outline of an evolutionary algorithm.

3. Evolutionary algorithm for input variable partitioning

The practical usefulness of functional decomposition for very complex systems is limited by lack of an efficient method for the construction of the high quality subsystems (*G* function from Fig. 1). In the subsystem construction process the following three factors play an extremely important role: an appropriate input support selection for subsystems, decision which (multi-valued) function will be computed by a certain subsystem and encoding of the subsystem's function with binary output variables. For function *F* of *n* input variables and the size *k* of input set of subsystem the number of possible solution is described by formula (20).

$$l = \binom{n}{k} = \frac{n!}{(n-k)!\,k!} \tag{20}$$

For large functions the solution space is so huge that heuristic method for solving this problem has to be used.

x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}	x_{11}	x_{12}	x_{13}
$\|U\| = 10$, $\|V\| = 3$, $\|\beta_G\| = 5$												
1	**1**	**1**	1	1	1	**0**	1	**1**	1	1	**0**	**0**
1	**1**	**1**	1	1	1	**0**	1	**1**	1	0	**1**	**0**
1	**1**	**1**	1	1	1	**0**	1	**1**	1	0	**0**	**1**
1	**1**	**1**	1	1	0	**0**	1	**1**	1	1	**0**	**1**
1	**1**	**1**	1	0	1	**0**	1	**1**	1	1	**0**	**1**
1	**1**	**1**	1	0	0	**1**	1	**1**	1	1	**0**	**1**
1	**1**	**1**	0	1	1	**0**	1	**1**	1	1	**1**	**0**
0	**1**	**1**	1	1	1	**1**	1	**1**	1	1	**0**	**0**
0	**1**	**1**	1	1	1	**1**	1	**1**	0	1	**1**	**0**
0	**1**	**1**	1	1	1	**1**	1	**1**	0	1	**0**	**1**
0	**1**	**1**	1	1	1	**1**	0	**1**	1	1	**1**	**0**
0	**1**	**1**	1	1	1	**1**	0	**1**	1	1	**0**	**1**
0	**1**	**1**	1	1	1	**1**	0	**1**	0	1	**1**	**1**
0	**1**	**1**	1	1	1	**0**	1	**1**	1	1	**1**	**0**
0	**1**	**1**	1	1	1	**0**	1	**1**	1	1	**0**	**1**
0	**1**	**1**	1	1	0	**1**	1	**1**	1	1	**0**	**1**
0	**1**	**1**	1	0	1	**1**	1	**1**	1	1	**0**	**1**
0	**1**	**1**	0	1	1	**1**	1	**1**	1	1	**1**	**0**
$\|U\| = 9$, $\|V\| = 4$, $\|\beta_G\| = 7$												
0	**1**	**1**	1	1	1	**0**	1	**1**	1	1	**0**	**0**
$\|U\| = 8$, $\|V\| = 5$, $\|\beta_G\| = 11$												
0	**1**	**1**	1	1	1	**0**	1	**1**	1	0	**0**	**0**
0	**1**	**1**	1	1	1	**0**	1	**1**	0	1	**0**	**0**
0	**1**	**1**	1	1	1	**0**	0	**1**	1	1	**0**	**0**
$\|U\| = 7$, $\|V\| = 6$, $\|\beta_G\| = 17$												
0	**1**	**1**	0	1	1	**0**	1	**1**	0	1	**0**	**0**
Frequency of appearance in *V* set.												
16	**0**	**0**	3	3	3	**13**	4	**0**	5	3	**16**	**13**

Table 2. Best input variable partitioning problem solutions of *plan* example.

The analysis of best possible solutions for given Boolean function results in interesting observations (Rawski, 2007b). Table 2 presents the best solutions of input variable partitioning for *plan* example Boolean function from standard Microelectronics Center of North Carolina benchmark set (Yang, 1991). This function has 13 inputs and 25 outputs.
Each row of Table 2 describes one partitioning of input variable set $X = \{x_1, \ldots, x_{13}\}$ into variables belonging to set U (marked by digit '1') and belonging to set V that leads to optimal decomposition (according to the number of blanket β_G's blocks). It presents the best solutions for different sizes of sets V and U, as well as the frequency of appearance of given input variable in V set. It can be easily noticed that certain variables appear in bound set often than others. For example variable x_1 appears in V set for 16 solutions listed in Table 2, while x_2 does not belong to V set for any of the best solutions. This suggests that some variables are more predestined to be included in V and other to be included in U set when constructing good input variable partitions.
There is another interesting observation that can be made analyzing this example. Let us assume that the size of V set is 4. Now, let us create an input variable partitioning in such way that V set consists of variables that according to Table 2 are least appropriate to be in bund set: $V = \{x_2, x_3, x_4, x_9\}$ and $U = \{x_1, x_5, x_6, x_7, x_8, x_{10}, x_{11}, x_{12}, x_{13}\}$. As we could expect, the quality of decomposition (according to the number of blanket β_G's blocks) is 16 – the worst possible for this size of V set. However let us move "good" variable x_1 from set U to set V and "bad" variable x_2 from set V to set U. The quality of decomposition is now 15, so it has improved. If we now swap variables x_3 and x_{12}, the decomposition will have quality 11, so further improvement has been obtained.
Let us assume that we have two variable partitioning solution (V_1, U_1) and (V_2, U_2). We can create another solution by taking part of variables from V_1 and part from V_2 and construct V_3 (similarly for U_3). Taking observation described above into account we can suspect that after such variable exchange it is probable that "good" variables from V_1 and V_2 will be included in V_3. This should improve the quality of new solution in comparison to solution used as "parents". If we preserve improved solutions and eliminate worsen solution we can apply this approach again. Such behavior is characteristic for evolutionary algorithms. This means that evolutionary algorithm may be an efficient way for solving input variable partitioning problem.
In (Rawski et al., 2004) the evolutionary algorithm has been proposed that solves input variable partitioning problem for functional decomposition. The evolutionary algorithm maintains a population of individuals (chromosomes), that represent potential solutions of a given optimization problem (Fig. 3). A survival of the fittest individuals is implemented by the selection mechanism. For the next population, as potential solutions, such single organisms are chosen, which adaptation to the environment is the best. The adaptation (quality) of a specific chromosome is evaluated by a fitness function. The chromosomes are evolving through the process of selection, recombination (crossover) and mutation. After a given number of algorithm loops (generations), it is expected that the algorithm has found a satisfactory solution. Details of the evolutionary algorithm solving input variable partitioning problem are discussed below.

3.1 Chromosome encoding

The single chromosome (organism) represents one, possible solution of the input variable partitioning problem. In the method presented in this paper chromosomes are encoded by the integer numbers, each of which represents the number of the input variable assigned to the set V (bound variables) of the decomposition.

Example 4.
For 4-input function F from Table 1 a possible solution of the variable partitioning problem can be represented by the set $U = \{x_1\}$ and set $V = \{x_2, x_3, x_3\}$.
The corresponding chromosome encoding is {2 3 4}.

3.2 Fitness function

In (Rawski et al., 1999) has been shown that there is a strong correlation of number of values in the sub-functions of the serial functional decomposition (represented by the number of blocks in β_G or size of *cut_set*) with the decomposition's quality. However this number strongly depends on the input variable partitioning chosen for the decomposition process. Therefore, the number of blocks in the β_G blanket or size of *cut_set* can be used as a good quality measure of the input variable partitioning. In the presented method the fitness function depends on this number – the less the number the better fitness of a given chromosome.
For the chromosome from Example 4 the number of blocks in a blanket β_G is $k = 3$ (Example 2).

3.3 Initial population selection

The initial population P^0 is created randomly. Once it is completed, the algorithm checks whether all the inputs (single genes) have been chosen at least once. If some are missing, the additional organism is created with genes which are not included in other organisms of the population.

3.4 Selection method

The selection method is combination of tournament selection and elitism. Tournament selection chooses randomly two organisms from the population P^t, compares them and takes the better one to the T^t population. The number of times such a tournament has to be done to complete whole T^t population depends on the population size. Elitism guarantees that the best organism from P^t is taken to T^t population regardless it was taking part at any tournaments or not.

3.5 Crossover (recombination)

Crossover operator chooses randomly two organism (called 'parents') and crosses their genetic material (Fig. 4). The crossover probability parameter specifies how often the crossover operator is performed. In proposed method this parameter is set to 0.9. The algorithm checks whether parents have the same genes or not. If so, the crossover operator is not launched and the other potential parents are chosen. If crossover is performed, two new organisms are created (and taken to O^t population). Otherwise parents are taken to O^t population.

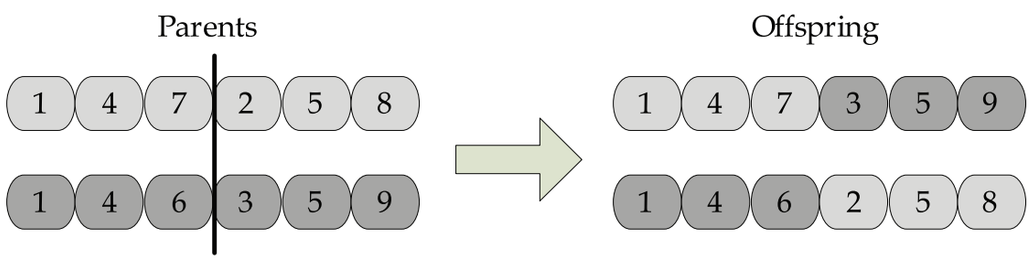


Fig. 4. Schematic representation of the crossover operator.

3.6 Mutation

Usually, mutation changes a single gene with very small probability (0.001). However, as experiments proved, in the case of the variable partitioning problem this kind of mutation does not bring any considerable profit for the algorithm performance.

The main problem with the presented algorithm is that it converges very fast to the local optimum. Once the algorithm gets to this area, it is very unlikely to find the better solution than this local optimum. To solve this problem, the special kind of mutation was implemented. If the average fitness among the population is very close to the best organism fitness, it is very likely the algorithm got stuck in the local optimum area. Then the special mutation is performed. One gene in each organism is mutated so the mutation probability is very high. As a result, the average fitness degenerates rapidly, but the algorithm gets out of the local optimum area and in many cases the better solution is found.

4. Input variable partitioning algorithm for heterogeneous LUTs

The methods presented in (Rawski et al., 2004), as well as in (Morawiecki & Rawski, 2008) were designed to solve the problem of input variable partitioning for given size of bound variable set V. In practice, during decomposition process there is a need to check existence of functional decomposition for several different sizes of V set before selecting the appropriate one. This is the case of applying functional decomposition in logic synthesis for FPGA architectures with heterogeneous logic resources. Such architectures are composed of adaptive logic elements that can be configured as LUTs of different sizes. In such situation application of concept presented in (Rawski et al., 2004) or (Morawiecki & Rawski, 2008) comes down to executing the algorithm for every possible LUT size.

However, the careful analysis of best possible solutions for *plan* example presented in Tab. 2 yields in interesting observation. Input variables that are present in bound set of best decompositions for set V of size k are often present in bound set of best decompositions for set V of size k – 1.

For example there is only one best solution for decomposition with set V of size 6 where $V = \{x_1, x_4, x_7, x_{10}, x_{12}, x_{13}\}$ and $U = \{x_2, x_3, x_5, x_6, x_8, x_9, x_{11}\}$. If we remove variable x_4 from set V and move it to set U we will obtain input variable partitioning that is one of 3 best solutions for decomposition with set V of size 5.

```
extended_evo_ivp( F , VSizeMin , VSizeMax )
begin
    V_VSizeMax := evo_ivp( F, VSizeMax )
    for k from VSizeMax– 1 downto VSizeMin do
    begin
        for i from 1 to k +1 do
        begin
            V_tmp := V_VSizeMax – { v_i }
            if (quality( V_tmp ) > best_quality) then
                V_best := V_tmp
                best_quality := quality( V_tmp )
        end
        V_k := V_best
    end
end
```

Fig. 5. Outline of algorithm that solves the problem of input variable partitioning for decomposition with V set of different sizes.

This can be used to construct the algorithm that applies evolutionary concept to find solution for largest size of set V only, and uses the found solution to construct solutions for decomposition with smaller set V. Let assume that we have evolutionary algorithm *evo_ivp*(F, *VSize*) that solves the problem of input variable partitioning for decomposition of function F with the bound set V of size *VSize*. To find good quality functional decompositions for V set of size from *VSizeMin* to *VSizeMax* the extended algorithm first will find solution (V set) for *VSizeMax* and then, by removing appropriate variables from found V set, it will construct solutions for decompositions with smaller sets V. The outline of the algorithm is presented on Fig. 5. The algorithm returns a list of V sets (V_k, where k = {*VSizeMin*, …, *VSizeMax*})

5. Results

The efficiency of evolutionary algorithm solving input variable partitioning problem for functional decomposition has been verified in (Rawski et al., 2004). This method was applied for number of combinational functions from MCNC logic synthesis benchmark set (Yang, 1991) and results ware compared with those obtained with the method based o information relationship measures presented in (Rawski et al., 2001) and with the systematic method. The systematic method is based on searching through the whole solution space and choosing an input support that produces blanket β_G with minimum possible number of blocks. All experiments were preformed on the computer with 512 Mbytes of RAM and AMD Athlon XP 3200.

Table 3 shows the comparison of the number of blanket β_G blocks for all methods for examples from MCNC logic synthesis benchmark set converted to truth table format (required by method from (Rawski et al., 2001). The results were obtained for decompositions with 3, 4, 5, and 6 input variables in set V. For these experiments the number of generations in the method based on the evolutionary algorithm was set to 30 and the size of a population was set to 40. Results obtained by the decomposition with the systematic search are optimal in the sense of the number of blocks of β_G. The method based on the evolutionary algorithm despite of its heuristic character produces results similar to the systematic method.

Table 4 present the minimum number of blocks of blanket β_G obtained by method based on the evolutionary algorithm for few large examples. The comparison with other two methods was impossible due to unacceptably long computation time for systematic method and due to the fact that method from (Rawski, 2001) accepts only truth table format. However examples from MCNC benchmark set are presented in espresso format, which in most cases is not truth table format and it is very difficult to convert such description for large multi-output systems into truth tables.

In Table 5 the comparison of execution time of the systematic method and the evolutionary-based algorithm is presented for different sizes of set V. As can be noticed, for large Boolean functions, method based on evolutionary algorithm is many times faster than the exact method. The difference in processing time between these two methods grows very fast with the function size. For the largest functions tried, the heuristic method was many thousands times faster.

	Size			Systematic method				Heuristic based on information relationship measures				Heuristic based on evolutionary algorithm			
	inputs	outputs	terms	3	4	5	6	3	4	5	6	3	4	5	6
Con1	7	2	20	5	6	6	5	5	7	7	5	5	6	6	5
Donfl	7	6	64	8	14	25	37	8	14	25	37	8	14	25	37
z4	7	4	128	4	6	8	12	4	6	8	12	4	6	8	12
Misex1	8	7	18	4	6	7	9	4	6	7	9	4	6	7	9
Root	8	5	71	5	9	15	17	5	9	15	17	5	9	15	17
Sqrt	8	4	53	3	4	7	12	3	4	7	12	3	4	7	12
Opus	9	10	23	4	6	8	10	4	6	8	10	4	6	8	10
9sym	9	1	191	4	5	6	7	4	5	6	7	4	5	6	7
Clip	9	5	430	6	10	14	18	6	10	14	21	6	10	14	18
Mark1	9	20	27	4	6	8	10	4	6	8	10	4	6	8	10
Alu2	10	3	391	6	12	24	43	6	12	24	43	6	12	24	43
Sao2	10	4	60	4	6	9	11	4	6	9	11	4	6	9	11
Cse	11	11	86	3	4	6	9	3	4	6	9	3	4	6	9
Sse	11	11	39	4	6	8	11	4	6	8	11	4	6	8	11
Keyb	12	7	147	6	9	13	19	6	9	13	19	6	9	13	19
S1	13	11	110	5	8	13	19	6	8	13	19	5	8	13	19
Plan	13	25	115	5	7	11	17	5	7	11	18	5	7	11	17
Styr	14	15	140	4	6	9	13	5	7	10	14	4	6	9	13
Ex1	14	24	127	4	6	8	11	4	6	8	11	4	6	8	11
Kirk	16	10	304	4	4	5	6	4	5	5	7	4	4	5	6
Duke2_7	18	1	64	3	4	4	4	4	5	5	5	3	4	4	4
Vg2_2	25	1	56	3	3	3	3	4	4	4	4	3	3	3	3
Apex3_3	34	1	208	2	3	4	5	4	4	4	6	2	3	4	5
Seq_2	36	1	211	2	2	3	–*)	3	4	4	6	2	2	3	5
Seq_1	37	1	286	2	3	3	3	3	3	3	4	2	3	3	3
Apex3_7	39	1	227	3	4	4	5	4	5	6	7	3	4	4	5

Table 3. Comparison of the number of blocks in blanket β_G obtained by the systematic method, heuristic method based on information relationship measures and heuristic method based on evolutionary algorithm for different size of set V. *) – too long computation time.

	Size			Heuristic based on evolutionary algorithm			
	inputs	outputs	terms	3	4	5	6
duke2	22	29	405	4	5	7	8
misex2	25	18	102	2	2	2	2
seq	41	35	3137	4	5	5	5
apex1	45	45	1440	4	5	6	7
apex3	54	50	1036	4	5	7	8
e64	65	65	327	4	5	5	7
apex5	117	88	2849	1	3	4	3

Table 4. The number of blocks in blanket β_G obtained by heuristic method based on evolutionary algorithm for different size of set V.

	Systematic method *				Heuristic based on evolutionary algorithm [s]			
	3	4	5	6	3	4	5	6
duke2	31s	2m 40s	10m 58s	34m 56s	37,7	38,9	40,4	58,5
misex2	10s	40s	4m 4s	13m 52s	10,2	12,4	14,9	24,9
seq	> 2 h	> 1 day	> 8 days	> 59 days	1656,8	1692,9	1704	1712,9
apex1	> 1 hour	> 12 hours	> 4 days	> 39 days	463,3	506,3	506	524,6
apex3	> 1 hour	> 16 hours	> 8 days	> 81 days	324	316,5	325	328,7
e64	31m 53s	> 8 hours	> 5 days	> 68 days	136,5	137,1	79	191,2
apex5	> 6 days	> 185 days	> 11 years	> 221 years	2849	3772,6	3799,5	3901,8

Table 5. Comparison of computation time of systematic method and heuristic method based on evolutionary algorithm for different sizes of set V.

In (Morawiecki & Rawski, 2008) the input selection method based on evolutionary algorithm was applied for decomposition based on BDDs. For manipulation on decision diagrams CUDD package was used. All the experiments were performed on the computer with 512 Mbytes of RAM and Pentium4 @ 2.8GHz.

	inputs	terms	Systematic method	Heuristic based on evolutionary algorithm
duke2_7	18	64	4	5
vg2_2	25	56	3	3
seq_2	36	211	2	4
apex1_16	38	179	2	3
apex1_23	39	2216	2	3

Table 6. The sizes of *cut_set* obtained by the systematic method and heuristic method based on evolutionary algorithm for size of set V equal to 4.

Table 6 presents the comparison of results obtained for single-output functions by applying the exhaustive search and evolutionary algorithm. The size of V set was 4 (typical value for FPGA-based synthesis). The size of *cut_set* is used as a quality measure. Results provided by the exhaustive search method can be considered as optimal. The results obtained by applying the evolutionary algorithm are very close to optimal or even optimal.

	Systematic method	Heuristic based on evolutionary algorithm [s]
duke2_7	80 sec	70
vg2_2	6,5 min	90
seq_2	31,5 min	75
apex1_16	46 min	120
apex1_23	49 min	130

Table 7. Comparison of computation time of systematic method and heuristic method based on evolutionary algorithm for sizes of set *V* equal to 4.

The comparison of execution time of the exhaustive search method and evolutionary algorithm has been presented in Table 7. It can be noticed that advantage of heuristic algorithm over exhaustive search grows fast with the size of decomposed Boolean function.

	Evolutionary algorithm				Extended evolutionary algorithm			
	3	4	5	6	3	4	5	6
duke2_7	4	4	5	5	4	4	4	5
vg2_2	3	3	3	3	3	3	3	4
seq_2	4	3	4	6	2	3	4	5
apex1_16	3	5	3	3	2	2	2	2
apex1_23	3	4	4	3	2	2	3	4

Table 8. Comparison of results of heuristic method based on evolutionary algorithm and its improved version.

Table 8 presents the comparison of results obtained with evolutionary algorithm and extended algorithm (Fig. 5) in case when existence of functional decomposition for several different sizes of *V* set has to be checked. It can be noticed that both methods provide results of comparable quality. However improved algorithm does it faster (Table 9). All the experiments were performed on the computer with 6 Gbytes of RAM and Intel Q9550 @ 2.83 GHz.

It has to be stressed that the multilevel decomposition consists of many single serial decomposition steps. Thus, application of the heuristic methods can speed up the multi-level decomposition process dramatically.

	Evolutionary algorithm [s]					Extended evolutionary algorithm [s]
	3	4	5	6	Total	Total
duke2_7	17.5	16.6	16.7	15.6	66.6	16.0
vg2_2	21.5	20.9	20.2	20.2	83.0	20.5
seq_2	15.9	16.0	15.1	16.1	63.3	16.6
apex1_16	17.1	16.6	15.7	14.6	64.2	16.0
apex1_23	17.1	15.2	15.1	15.4	63.0	15.5

Table 9. Comparison of computation time of heuristic method based on evolutionary algorithm and its improved version.

6. Conclusion

The heuristic method of variable partitioning based on the evolutionary algorithm turns out to be very efficient when applied for decomposition method based on cubes (Rawski, 2007a), as well on ROBDDs (Morawiecki & Rawski, 2008). The method delivers results of similar or comparable quality to results obtained from the exhaustive search, but does it many times faster. The algorithm parameters (the number of generations and the size of population) can be used to control the trade-off between the search time and quality of solutions. These features make the proposed heuristic method very useful for decomposition-based synthesis of large systems.

7. Acknowledgments

This work was partly supported by the Ministry of Science and Higher Education of Poland – research grant no. N N516 418538 for 2010-2012.

8. References

Akers, S. B. (1978). Binary decision diagrams. *IEEE Transactions on Computers*, Vol. 27, No. 6, pp. 509–516.

Bryant, R. E. (1986). Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, Vol. 35, No. 6, pp. 677–691.

Brzozowski, J. A. & Łuba, T. (2003). Decomposition of Boolean Functions Specified by Cubes, *Journal of Multiple-Valued Logic and Soft Computing*. Vol. 9, pp. 377–417.

Holland, J.H. (1975). Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence. MIT Press.

Khuri, S. (1994). An Evolutionary Approach to Combinatorial Optimization Problems. *Proceedings of CSC*, pp. 66 – 73.

Lee, C. Y. (1959) Representation of switching circuits by binary-decision diagrams. *The Bell System Technical Journal*, pp. 985–999.

Łuba, T.; Selvaraj, H.; Nowicka, M. & Kraśniewski, A. (1995). Balanced multilevel decomposition and its applications in FPGA-based synthesis. G.Saucier, A.Mignotte (ed.), *Logic and Architecture Synthesis*, Chapman&Hall.

Łuba, T. & Selvaraj, H. (1995). A general approach to Boolean function decomposition and its applications in FPGA-based synthesis, *VLSI Design*, Vol. 3, No. 3-4, pp. 289–300.

Morawiecki, P. & Rawski M. (2008) Method of Input Variable Partitioning in Functional Decomposition Based on Evolutionary Algorithm and Binary Decision Diagrams, *Proc. of IEEE 2008 Conference Human Systems Interaction*, publication on CD,.

Noviskey, M.J.; Ross, T.D.; Gadd, D.A. & Axtell M. (1994). Application of Genetic Algorithms to Functional Decomposition in Pattern Theory. Report WL-TR-94-1015, Wright Laboratory, WL/AART-2. WPAFB, OH 4533-5543.

Perkowski, M. (1994) A Survey of Literature on Function Decomposition. *Final Report for Summer Faculty Research Program*. Wright Laboratory. Sponsored by Air Force Office of Scientific Research. Bolling Air Force Base. DC and Wright Laboratory.

Perkowski, M; et al. (1997). Decomposition of Multiple-Valued Relations. *International Symposium on Multiple-Valued Logic*, pp. 13-18.

Rawski, M.; Selvaraj, H & Morawiecki, P. (2004) Efficient Method of Input Variable Partitioning in Functional Decomposition Based on Evolutionary Algorithms, *Proc. Euromicro Symposium on Digital System Design, Architectures, Methods and Tools*, IEEE Computer Society, Selvaraj H. (Editor), pp. 136-143.

Rawski, M. (2007a). Decomposition of Boolean function sets, *Electronics and Telecommunications Quarterly*, Vol. 53, No. 3, pp. 231–249.

Rawski, M. (2007b). Efficient Variable Partitioning Method for Functional Decomposition, *Electronics and Telecommunications Quarterly*, Vol. 53, No. 1, pp. 63-81.

Rawski, M.; Jóźwiak, L. & Łuba, T. (2001). Efficient Input Support Selection for Sub-functions in Functional Decomposition Based on Information Relationship Measures. *Journal of Systems Architecture*, Vol. 47 , No. 2, p. 137-155.

Rawski, M.; Jóźwiak, L. & Łuba, T. (1999). The Influence of the Number of Values in Sub-Functions on the Effectiveness and Efficiency of Functional Decomposition. *Proceedings of the 25th EUROMICRO Conference*, IEEE Computer Society, pp. 86-93.

Rawski, M.; Tomaszewicz, P.; Selvaraj, H. & Łuba T. (2005). Efficient implementation of digital filters with use of advanced synthesis methods targeted FPGA architectures, *Proceedings of Euromicro Symposium on Digital System Design, Architectures, Methods and Tools*, IEEE Computer Society, Wolinski C. (Editor), pp. 460–466.

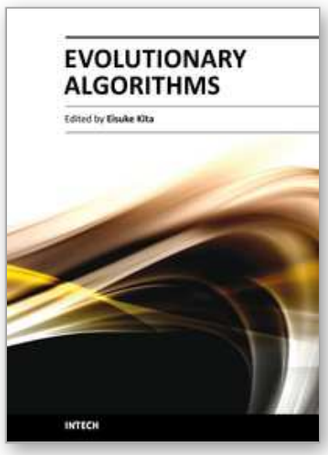
Sasao, T.; Matsuura, M.; Iguchi, Y. & Nagayama, S. (2001). Compact BDD Representations for Multiple-Output Functions and Their Application, *Proceedings of ISMVL,* pp. 207-212.

Scholl, C. (2001). *Functional Decomposition with Application to FPGA Synthesis*. Kluwer Academic Publisher.

Steinbach, B. & Stokert, M. (1994). Design of Fully Testable Circuits by Functional Decomposition & Implicit Test Pattern Generation. *Proceedings of IEEE VLSI Test Symposium*, pp. 22–27

Wurth, B.; Schlichtmann, U.; Eckl, K. & Antreich, K. (1999). Functional multiple-output decomposition with application to technology mapping for lookup table-based FPGAs. *ACM Trans. Design Autom. Electr. Syst.* Vol. 4, No. 3, pp. 313-350.

Yang, S. (1991). Logic Synthesis and Optimization Benchmarks, Version 3.0. *Tech. Report*, Microelectronics Center of North Carolina.



Evolutionary Algorithms

Edited by Prof. Eisuke Kita

ISBN 978-953-307-171-8
Hard cover, 584 pages
Publisher InTech
Published online 26, April, 2011
Published in print edition April, 2011

Evolutionary algorithms are successively applied to wide optimization problems in the engineering, marketing, operations research, and social science, such as include scheduling, genetics, material selection, structural design and so on. Apart from mathematical optimization problems, evolutionary algorithms have also been used as an experimental framework within biological evolution and natural selection in the field of artificial life.

InTech Europe
University Campus STeP Ri
Slavka Krautzeka 83/A
51000 Rijeka, Croatia
Phone: +385 (51) 770 447
Fax: +385 (51) 686 166
www.intechopen.com

InTech China
Unit 405, Office Block, Hotel Equatorial Shanghai
No.65, Yan An Road (West), Shanghai, 200040, China
中国上海市延安西路65号上海国际贵都大饭店办公楼405单元
Phone: +86-21-62489820
Fax: +86-21-62489821

IntechOpen