A New Decomposition Algorithm for Threshold Synthesis and Generalization of Boolean Functions

José L. Subirats, José M. Jerez, and Leonardo Franco, *Member, IEEE*

***Abstract*—A new algorithm for obtaining efficient architectures composed of threshold gates that implement arbitrary Boolean functions is introduced. The method reduces the complexity of a given target function by splitting the function according to the variable with the highest influence. The procedure is iteratively applied until a set of threshold functions is obtained, leading to reduced depth architectures, in which the obtained threshold functions form the nodes and a AND or OR function is the output of the architecture. The algorithm is tested on a large set of benchmark functions and the results compared to previous existing solutions, showing a considerable reduction on the number of gates and levels of the obtained architectures. An extension of the method for partially defined functions is also introduced and the generalization ability of the method is analyzed.**

***Index Terms*—Circuit complexity, generalization, linear separability, logic synthesis, threshold networks.**

I. Introduction

In this paper we introduce an algorithm for the decomposition of a given Boolean function in a set of linearly separable (or threshold) functions permitting to obtain small size circuit architectures. This problem is known as the threshold logic synthesis of Boolean functions and has been much studied since the 1960s [1]–[5]. The interest in the area of circuit design in using threshold gates (or linearly separable functions) instead of standard Boolean logic AND, NOT, and OR gates relies on the fact that threshold elements are more powerful, in the sense that the size of the circuits that can be constructed to compute the desired functions can be smaller [6]–[8]. There is an extra interest in the study of threshold circuits because networks constructed with threshold gates are almost equivalent to standard feedforward neural networks models using sigmoidal activation functions, and, thus, most of the properties and characteristics of the circuits can be extended and applied to neural networks [9]–[11]. The standard practice for the architecture selection process within the field of artificial neural networks is the trial-and-error method that is not very efficient. To avoid the drawbacks of the architecture selection process and training procedure different constructive algorithms have been proposed [12]–[15]. A comparison and evaluation of some of these algorithms can be found in [16], [17]. The problem of finding efficient architectures is relevant also from a more theoretical point of view within the area of circuit complexity where the minimum size circuits needed to implement functions are analyzed, because existing bounds can be checked or improved [18]–[22]. Different threshold circuits have been implemented in hardware (see [23] for a review) but the commercial application of threshold circuits is still in its infancy. More recently, due to advances on the field of nanotechnology, there has been some renewed interest on methods to synthesize architectures with threshold gates as the construction of these circuits has become feasible and might be a real alternative to standard circuits [24]–[26].

An extra motivation of the present paper is the study of (quasi)-optimal neural architectures that implement a given Boolean function and also the analysis of the generalization ability of the generated networks. Recent synthesis algorithms that include results related to those presented in this paper includes the works by Zhang *et al.*, (2005) [25] and Avedillo and Quintana, (2004) [24]. Zhang *et al.*, (2005) [25] implemented a system to build threshold circuits for Boolean functions. They use as starting point the output of the SIS system [29] that gives a standard Boolean circuit computing the desired function. Their algorithm then tries to reduce the number of gates by replacing the Boolean gates by threshold gates using a node collapsing algorithm. Avedillo and Quintana [24] used an exhaustive architecture search tool for the construction of threshold networks. Exhaustive results can be only applied to limited size circuits due to the exponential complexity of the problem and thus they applied their method to Boolean circuits with a fan-in max of seven generated from the SIS system. The main difference of our implementation with both aforementioned approaches is that our algorithm works directly with the truth table of the Boolean function, and, thus, all techniques for function splitting and checking linearly separability are optimized for this type of representation. The advantages and disadvantages of the different implementations are analyzed in the discussion where we also compare the results obtained. Other existing approaches related to circuit synthesis and generalization include the works in [30]–[32], where different methods for binary classification tasks have been implemented.

The organization of the present paper is as follows. In Section II, the mathematical concepts needed to introduce the algorithm are presented, followed by Section III where the algorithm itself is introduced. In Section IV the results obtained from the application of the algorithm to a set of benchmark Boolean functions are presented. First, results on the synthesis of restricted size functions of up to 21 inputs are shown, where

Manuscript received April 3, 2007; revised October 11, 2007. First published April 18, 2008; current version published November 21, 2008. This work was supported by Grant CICYT-TIN2005-02984 (including FEDER funds) and Grant P06-TIC-01615. The work of L. Franco was supported by the Spanish Ministry of Education and Science through a Ramón y Cajal fellowship. This paper was recommended by Associate Editor K. Chakrabarty.

The authors are with the Departamento de Lenguajes y Ciencias de la Computación, Universidad de Málaga, 29071 Málaga, Spain (e-mail: lfranco@lcc.uma.es).

Digital Object Identifier 10.1109/TCSI.2008.923432

the decomposition algorithm for synthesis and generalization (DASG) algorithm is applied directly to the whole function truth table. Later, in Section IV-B, the results for larger size functions of up to 257 inputs are presented where the DASG algorithm is applied after a preprocessing stage done by using the SIS system. The new algorithm is also extended to the case of partially defined functions in Section V, where we also apply it to measure the generalization ability of the new algorithm on a set of benchmark functions. The paper ends with a discussion and conclusion of the results and the new methods introduced in this work.

II. Mathematical Preliminaries

A Boolean function of n variables is defined as a mapping $f: \{0,1\}^n \rightarrow \{0,1\}$. A Boolean function is completely specified by the output of the function, $f(x)$, in response to each of the input examples, which can be represented by a vector, the truth vector of the function $t = [t_0, t_1, \ldots, t_{2^n-1}]$.

The Hamming weight H of a Boolean function is equal to the number of elements of the truth vector equal to 1.

A threshold gate of n input variables $x_i \in \{0,1\}$ has n weights, w_i, associated to the variables and a threshold, T. The function computed by the threshold gate is a function of the weights and the threshold, such that for each input vector, $x = (x_0, \ldots, x_{n-1})$, the value of a function f is

$$f(x_0, \ldots, x_{n-1}) = \begin{cases} 1 & \text{if } \sum_{i=0}^{n-1} w_i\, x_i - T \geq 0 \\ 0 & \text{otherwise} \end{cases}.$$

Any function that can be computed by a threshold gate is called a threshold function or equivalently a linearly separable Boolean function.

A symmetric Boolean function is a function whose truth vector depends only on the Hamming weight of the input (any permutation of the input variables does not produce a change of output).

The negation of a variable, x_i, written as: $\bar{x}_i$ consists in changing its value from 0 to 1 or vice versa (i.e., the transformation $\bar{x}_i = 1 - x_i$). A function $f(x_1, \ldots, x_n)$ is positive in x_i if and only if $f(x_0, \ldots, x_i = 0, \ldots, x_{n-1}) \leq f(x_0, \ldots, x_i = 1, \ldots, x_{n-1})$. Similarly, a function $f(x_1, \ldots, x_n)$ is negative in x_i if and only if $f(x_0, \ldots, x_i = 0, \ldots, x_{n-1}) \geq f(x_0, \ldots, x_i = 1, \ldots, x_{n-1})$. A Boolean function is said to be unate in a variable x_i if and only if f is positive or negative in x_i. If f is unate in all of its variables, then f is simply said to be unate.

The influence of a variable x_i is defined as the fraction of inputs vectors $x \in \{0,1\}^n$ such that a change of the value of the variable causes a change on the value of the function. To clarify, if an input vector $(x_1, \ldots, x_i, \ldots, x_n)$ is modified by negating variable x_i to get the new vector $(x_1, \ldots, \bar{x}_i, \ldots, x_n)$, the influence of the variable measures how many times out of the total existing cases, a change on that variable produces a change on the output of the function. An example of the calculation of influence is given in Section III-D.

The self-dual extension of a given Boolean function of n variables is the $n+1$ dimensional function defined as

$$f^{n+1}(x_0, \ldots, x_{n-1}, x_n) = \begin{cases} f^n(x_0, \ldots, x_{n-1}) & \text{if } x_n = 0 \\ 1 - f^n(\bar{x}_0, \ldots, \bar{x}_{n-1}) & \text{if } x_n = 1 \end{cases}.$$

III. The New DASG Algorithm for the Synthesis of Boolean Functions

We introduce here a new algorithm named DASG, that permits to find for an arbitrary given target Boolean function an architecture comprising threshold gates that will compute the desired function. The algorithm reduces the complexity of the original function by splitting it, according to the most convenient variable, to create two new less complex functions; the procedure is repeated until the obtained functions are all linearly separable. The method of splitting a function in terms of a given variable is known as Shannon decomposition and it was formally introduced in 1938 by Shannon. Decomposition algorithms are a classical research subject of switching theory and have been the main path to multiple level logic synthesis in the 1960s [33], [34].

With the help of a diagram (Fig. 1), we first present an overview of the whole algorithm and then in separate subsections, technical details of the different steps applied are given. To understand the steps taking in the process, we will follow the scheme shown in Fig. 1. First, the target function (TF) is entered, and according to its Hamming weight, it is decided whether to use an OR or AND representation of the function. The choice of representation affects the algorithm in two ways: first, the output function of the final architecture will be an OR (AND) if more than half of the outputs of the target function are 1's (0's). Second, it affects the function splitting procedure, as 0's or 1's will be used for filling new undefined instances created after the splitting process is applied. For the case of an input target function with half of the bits 0, any of the two choices can be used. After the representation to be used is selected, the whole decomposition procedure continues by adding the target function to the work-set, a reservoir that contains all functions that will be analyzed. At the beginning, the work-set contains only the target function but later on it will contain the newly created functions after the splitting procedure is applied iteratively. The algorithm continues by picking a function from the work-set and applying a variable elimination procedure (indicated by "simplify f." in the diagram of Fig. 1) to eliminate irrelevant variables which have no influence in the output of the function. Afterwards, a linear separability test is applied in two steps; first the unateness of the function is tested in the self-dual space, and if the function is unate in this $n+1$ dimensional space the linear separability is tested by means of a linear programming code. If the function has not passed any of the tests, then is nonlinearly separable and the functions are added to the work-set. Otherwise, if the function under analysis is linear separable it is added to the solution set. The whole procedure is repeated until there are no more functions to be analyzed in the work-set. The set of found threshold functions

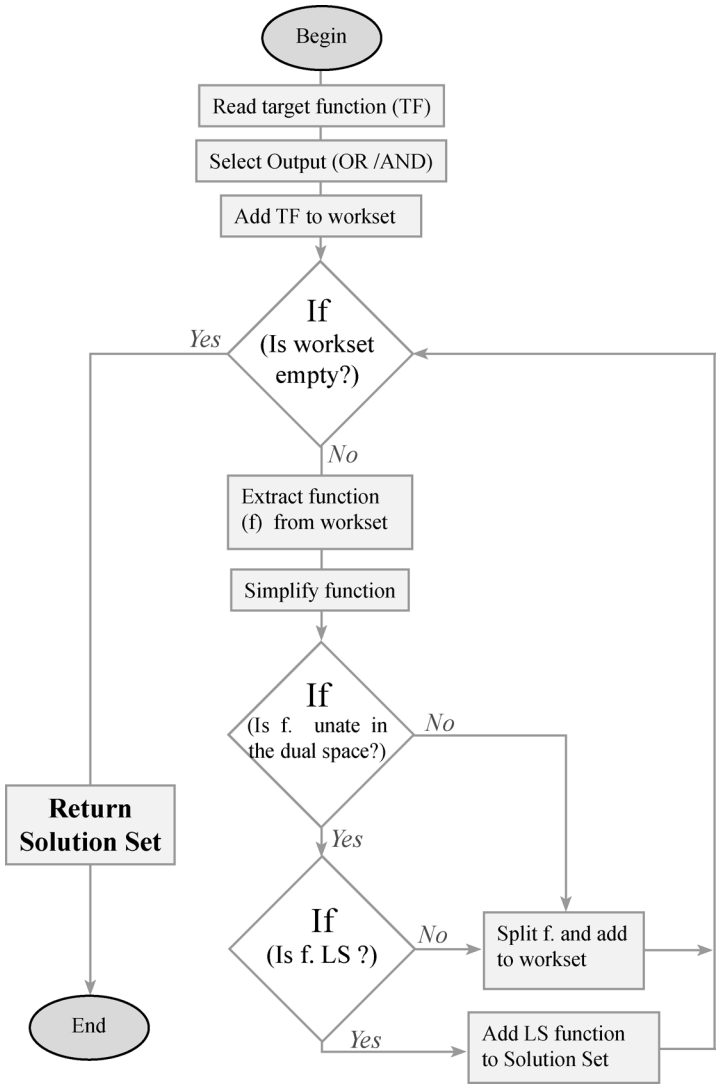

Fig. 1. A scheme of the DASG algorithm for decomposing a given target function in a set of linearly separable (LS) functions.

are then all combined using the OR or AND function selected at the beginning, and the final output will compute the desired function.

A. The Importance of the Weight of the Target Function in the Selection of the Output

The algorithm introduced above uses as output function of the whole circuit an AND or OR function depending on the Hamming weight of the target function. The reason for having two choices lies on results that we obtained analyzing optimal architectures, that indicates that good choices for output functions are ones with a similar Hamming weight as the target function [35]. We made simulations of the algorithm in which both the OR and AND representations were tested on the same function and the results showed that smaller architectures were obtained when the output function is selected according to the criteria stated above.

B. The Tests for Linear Separability

At each stage, the algorithm needs to check for the linear separability of the functions found, and thus it is desirable that the method used would be as fast and efficient as possible. All threshold functions are unate functions and then a first test could be to check whether the obtained functions are unate. Franco *et al.* (2006) [36] have improved the test by demonstrating (See theorem [36, Section III]) that checking unateness in the self-dual space of $n+1$ variables is a more stringent test than checking unateness in the original n dimensional space. Nevertheless, the test is not exhaustive, and thus, in order to eliminate the possibility of using a nonthreshold function, a second procedure is applied. The test is based on an integer linear programming method based on the simplex approximation, and we used the version implemented in a public available and well-known package LP SOLVE [37], that also gives the values of the weights and threshold needed to compute the function.

C. Function Decomposition Based on Variable Influence

When the algorithm finds a function from the work-set that is not threshold, the procedure continues by splitting the function using the nonunate variable with the highest influence. The splitting creates two functions $F1_a$ and $F1_b$ that are obtained

according to the following set of equations for the case of using an OR representation for the solution:

$$F1_a(x_0,\ldots,x_i,\ldots,x_{n-1}) = \begin{cases} F(x_0,\ldots,x_i=0,\ldots,x_{n-1}) & \text{if } x_i=0 \\ 0 & \text{if } x_i=1 \end{cases} \quad (1)$$

$$F1_b(x_0,\ldots,x_i,\ldots,x_{n-1}) = \begin{cases} 0 & \text{if } x_i=0 \\ F(x_0,\ldots,x_i=1,\ldots,x_{n-1}) & \text{if } x_i=1 \end{cases}. \quad (2)$$

The splitting equations for the case of using the AND representation are similar but the undetermined outputs are set to 1 instead of 0.

We tested different strategies for the splitting procedure as it is critical for the success of the whole algorithm, finding that the more efficient strategy was to use the variable with the highest influence among the nonunate variables. We carried a thorough comparison of the results for the case of using two different alternatives. First, we tested the procedure when the splitting variable was chosen at random. This option lead to an average increase of 46.1% per function on the number of gates and a 63.7% increase on the number of weights, while the number of levels is not affected. The second tested alternative was to split the function using the variable within the nonunate ones with the largest value of positive or negative influence. For this last case, the results show an average increase per function of 16.1% in the number of gates of the generated architectures and a 19.2% increase in the interconnect per function.

Note that it may happen that a function from the work-set is unate in the $n+1$ dimensions self-dual space but not threshold. For this case, the splitting procedure can be applied in a similar way, but selecting the splitting variable as the one with the largest influence among all variables.

In Zhang *et al.* [25], they use an splitting technique based on choosing the most frequently appearing variable on the algebraic form of the Boolean function. We have checked and compared both approaches and find that they are not equivalent and thus lead to different results.

D. An Example of the Application of the Algorithm to the Synthesis of a 4 Variables Function

As an example of how the algorithm works, we apply it to the decomposition of a 4-input function F, with truth vector {1,0,0,1,1,0,1,1,1,1,1,1,0,1,0,1}. A sum of products (SOP) representation of the function is the following: $F = x_0x_3 + x_0x_1\bar{x_3} + \bar{x_0}\bar{x_1}\bar{x_2} + \bar{x_0}x_2\bar{x_3} + \bar{x_0}x_1\bar{x_2}x_3$.

The function is not unate and thus is not a threshold function. The whole decomposition procedure for this function is shown in Fig. 2. On top of the figure, the truth vector of the target function is shown. The weight of the target function is 11, i.e., more than half of the output bits are 1, and, thus, an OR representation for the decomposition will be used. To decompose the function, first, the unateness of the 4 variables is analyzed and for the nonunate ones the influence, S_{xi} is computed. The influence of a variable x_i is the fraction of inputs, such that a negation of the variable produces a change on the output of the function. For example, for the function we are considering, the value of the influence for the second variable x_1 is $6/16 = 3/8$,

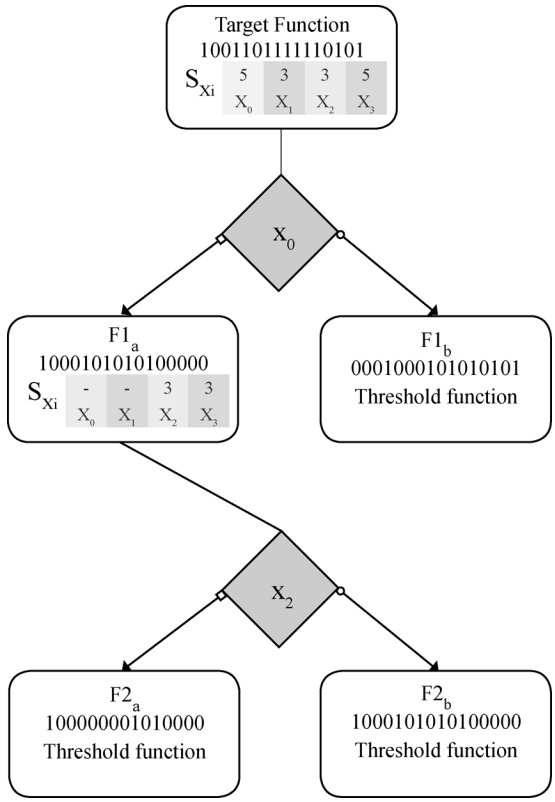

Fig. 2. An example of the DASG algorithm applied to a Boolean function of four input variables with truth vector {1,0,0,1,1,0,1,1,1,1,1,1,0,1,0,1}. The final obtained architecture is depicted in Fig. 3.

as the input vectors 0010, 1000, and 1010 when modified by changing the value of x_1 (the second variable from left to right) from 0 to 1 produce inputs whose output differs from the original one. In this work, in order to simplify the diagrams and not use fractional values, the influences values are multiplied by 2^{N-1} (the method depends on the relative values of the influences and thus this re-scaling of values does not alter the results). Another advantage of using this transformation, is that the value of the scaled influences for a variable i is simply the number of pairs of input vectors with different output that differ only in the value of variable i: in the example, the 3 pairs of inputs 0010–0110, 1000–1100 and 1010–1110 have opposite outputs and differ on the second input variable x_1.

In the top box of Fig. 2, the values of the scaled influences for the four nonunate variables are shown . The influence is higher and has a value of 5 for variables x_0 and x_3. In this case, any of the two variables can be selected for splitting the function but for the implementation of the algorithm the first variable with the highest influence is selected (in the example, the variable selected is x_0). To split the function, the formulas of (1)–(2) are used. The newly created functions, $F1_a$ and $F1_b$ will contain half of the bits inherited from the original function and the other half are equal to 0. The procedure continues by checking whether the new functions are threshold functions or not. In the example, one of the new created functions, indicated by $F1_b$ is a threshold function and thus added to the solution set. The other function, $F1_a$, is not linearly separable, and, thus, the splitting procedure is applied another time. In the picture, the influences

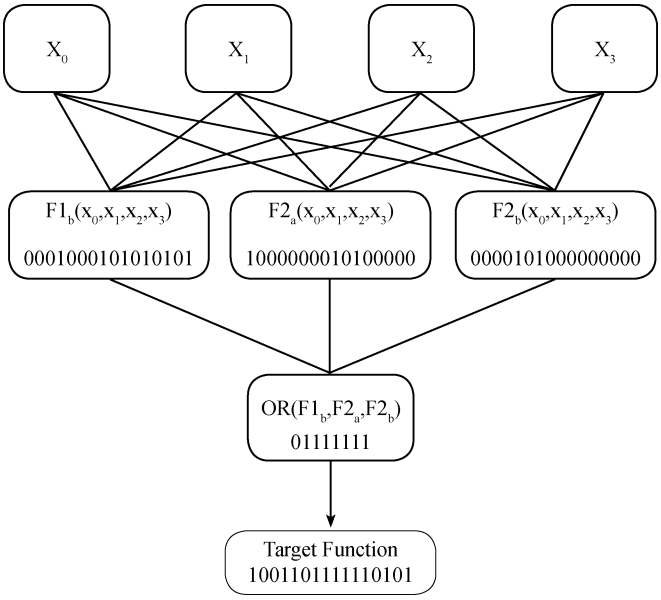

Fig. 3. The final solution found with the algorithm for a test function of four variables used to show how the DASG algorithm works. The procedure used to arrive to the solution is shown in Fig. 2.

of the different variables are shown, and a hyphen ('-') is used to indicate that a variable is unate and then is not a candidate variable for the splitting. The function $F1_a$ is then split into two new functions, using the variable x_2. Finally, the two obtained functions ($F2_a$, $F2_b$) are linearly separable and the decomposition procedure ends. The final architecture for computing the target function is shown in Fig. 3.

IV. Performance of the Algorithm on a Set of Benchmark Functions

We tested the algorithm on a set of widely used circuit functions belonging to the MCNC benchmark with a number of variables between 3 and 257. We analyze the performance of the new DASG algorithm in two parts. In next subsection, we show the results of applying the algorithm directly to the whole truth table of functions of up to 21 inputs and later on Section IV-B, the algorithm is applied to larger functions of up to 257 inputs after the application of some pre-processing using the SIS program suite that reduces the fan-in of the functions.

A. Results for Functions of Up to 21 Inputs Applying the Algorithm Directly to the Truth Table

The main features of the obtained architectures are shown in Table I, where the results are compared to those from [25]. In the table, the total number of gates (G), the number of levels (L), the interconnect (number of weights, W), and the fan-in max (F) are shown for each of the functions. The last three columns of the Table show the percentage of improvement (reduction) in the number of gates, number of connections and number of levels in comparison to the algorithm by Zhang *et al.* [25] (negative values indicate that the results from the DASG algorithm show an increase with respect to the ones obtained by [25]).

The DASG algorithm when applied directly to the definition of the whole function produces circuits with only one hidden layer of threshold gates (or neurons), that in certain cases can

TABLE I
The Results Obtained for the Synthesis of Architectures for 24 Multi-Output Boolean Function Using the DASG Algorithm. The Columns Indicate the Name and the Number of Inputs and Outputs of the Function, the Number of Gates (G), the Number of Levels (L), the Total Number of Weights (W), and the Fan-In Max (F) of the Generated Circuits. In the Last Three Columns the Percentage of Improvement Over the Results Reported in [25] are Shown

		DASG				% Reduction		
	I/O	G	L	W	F	G	L	W
b1	3-4	7	2	16	3	12	33	0
cm42a	4-10	10	1	40	4	23	67	-18
decod	5-16	16	1	80	5	33	67	-54
cm82a	5- 3	11	2	45	5	8	50	-18
majority	5-1	1	1	5	5	0	50	0
z4ml	7-4	16	2	79	7	16	60	-23
f51m	8-8	32	2	153	4	61	75	42
9symml	9-1	21	2	200	20	81	78	51
alu2	10-6	96	2	822	46	51	92	-15
x2	10-7	17	2	79	7	-13	50	-18
cm152a	11-1	9	2	40	8	18	50	5
cm85a	11-3	19	2	180	16	-36	60	-150
cm151a	12-2	18	2	96	8	-50	60	-113
alu4	14-8	279	2	3189	86	32	91	-127
cm162a	14-5	15	2	72	10	42	75	18
cu	14-11	22	2	122	10	8	50	-60
cm163a	16-5	21	2	105	8	16	67	-25
cmb	16-4	71	4	48	12	85	83	32
pm1	16-13	17	2	77	9	26	50	-1
tcon	17-16	24	2	56	2	25	33	0
pcle	19-9	32	2	166	11	8	67	-52
sct	19-15	23	2	134	13	39	60	-16
cm150a	21-1	17	2	112	16	19	50	-45
cc	21-20	44	2	189	6	-26	67	-108
Average		32.1	1.8	254	13	20	62	-29

be modified to a large number of levels if a reduction of fan-in is required. The fan-in max of the generated circuits is generally equal to the number of relevant input variables, i.e., variables that have influence on the output of the function. But, if the number of threshold functions forming the hidden layer, in which the target function has been decomposed, is larger than the number of relevant input variables the output neuron will have a fan-in equal to this number of gates. However, in this case a simple transformation can reduce the fan-in max down to the number of input relevant variables. This transformation is possible because the output neuron of the architectures is an OR or AND gate.

The computing time needed for the algorithm to operate was analyzed for some single output functions. For all functions tested with a number of variables less than 11 the computational time needed was less than 2 s, but when the number of variables increases the computational cost grows considerably, both due to the need of computing power and memory resources. For example, for the three single output functions *pm1c0*, *sctd0* and *cm150av* with 16, 19, and 21, variables, respectively, the time needed to obtain the final solution was 28, 2382, and 49 383 s, and, thus, we restricted the set of analyzed functions to those having a maximum of 21 variables. The computer used for the test is a standard PC Pentium 4, 3.2 Ghz with 1 GB of RAM memory running *C#* under Windows. We excluded from our analysis the parity function on 16 variables used in [25]. The parity function [6], [38] is a well known and very complex symmetric function and is the worst possible case for our algorithm. The DASG algorithm needs very long computational times for

processing the parity function as it splits the function on every single variable, creating exponentially large circuits. It is clear from the way that the DASG algorithm works that the symmetry of the functions is not taken into account, and this makes the DASG algorithm inefficient for symmetric functions. Fortunately, the set of symmetric Boolean functions is a very small set comprising only a tiny fraction of the whole set of functions and it is also known that architectures with a single hidden layer and n neurons are able to implement any symmetric function.

In the last row of Table I, the average obtained results are shown. It can seen be that the DASG algorithm outperforms the results obtained by Zhang *et al.* [25], in terms of the number of gates and levels used in the circuit. On average a reduction of 20% in the total number of gates was observed, and for certain cases the reduction on the number of gates went up to 85%. It is worth noting that the results of [25] showed already a large improvement on the size of the circuits obtained in comparison to those obtained directly by one-to-one mapping of circuits composed of standard Boolean gates. For this set of functions, the DASG algorithm creates architectures with only two levels but the generated architectures have a larger number of connections in comparison to the ones obtained by Zhang *et al.*, as the average interconnect was 28% larger. This increase shows a trade off between the number of connections and the number of gates and levels of the constructed architectures. However, it is worth noting that in 5 out of the 24 multi-output analyzed functions (f51m,9symml,cm152a,cm162a and cmb), a reduction in all three features simultaneously was obtained.

B. Results for Functions Between 23 and 257 Inputs

The introduced DASG method can only be applied directly to functions specified by its truth table when the number of input variables is less or equal to 21. This limitation is due to cpu and memory requirements when the number of input variables, N, increases above 21, as the number of instances that have to be analyzed is 2^N. In order to apply the DASG algorithm to functions with a larger number of inputs it is necessary to use functions specified by relationships between the input variables, as it is the standard procedure in circuit design. Thus, we applied some standard processing using the program SIS [29]. First, the script named *script.rugged* was applied, later the command *map -m 0*, to then iteratively use the command *reduce_depth* to reduce the number of levels while at the same time the maximum fan-in permitted was increased. The results reported below are those obtained by applying the DASG algorithm after some iterations of the above mentioned SIS commands for which an increasing in the fan-in permitted produces a reduction on the number of gates. The results are shown in Table II for functions with a number of inputs between 23 and 257. In the table, the features of the generated architectures obtained are compared to the ones obtained by Zhang *et al.* Similar results to those obtained previously with functions of up to 21 inputs (shown in Table I) can be seen, as a reduction in the number of gates and levels was obtained while some increase in the interconnect is observed. The reduction in the number of gates was of 7.9%, a bit lower than in the case of applying the DASG algorithm directly to the truth table of the whole function where a 20% reduction was obtained. The reduction in the number of levels was still considerably high (41%), with architectures having on average a number of 5.5 levels. The average maximum fan-in of the obtained architectures was 7.9 with values in the range between 4 and 15.

TABLE II
THE RESULTS OBTAINED FOR THE SYNTHESIS OF 31 MULTI-OUTPUT BOOLEAN FUNCTIONS WITH A NUMBER OF INPUTS BETWEEN 23 AND 257. THE DASG ALGORITHM WAS APPLIED TO PARTIAL FUNCTIONS OBTAINED AFTER APPLYING SOME TRANSFORMATION COMMANDS USING THE SIS SYSTEM. (THE NOTATION AND THE FEATURES REPORTED IN THE TABLE ARE SIMILAR TO THOSE USED IN TABLE I

		DASG				% Reduction		
	I/O	G	L	W	f	G	L	W
cordic	23-2	33	7	130	8	33	-17	16
ttt2	24-21	77	2	462	14	23	67	-264
i1	25-16	21	3	69	7	9	40	-10
lal	26-19	51	3	186	8	6	57	-11
pcler8	27-17	40	3	124	13	15	57	13
frg1	28-3	59	5	207	9	0	44	11
c8	28-18	67	2	216	13	21	71	5
comp	32-3	52	5	208	7	37	38	33
my adder	33-17	89	9	275	6	7	50	10
term1	34-10	90	6	318	8	60	40	53
count	35-16	57	5	238	9	28	58	1
unreg	36-16	64	2	208	4	-28	60	-55
cht	47-36	110	3	352	5	32	40	-74
apex7	49-37	126	6	496	4	42	33	-12
x1	51-35	138	4	501	4	8	43	31
dalu	75-16	571	7	2421	8	30	70	6
example2	85-66	186	6	520	7	-2	25	-6
i9	88-63	437	7	1242	6	-59	13	-52
x4	94-71	216	7	729	12	-14	13	-30
i3	132-6	102	3	166	8	35	50	30
i5	133-66	82	4	134	7	-24	33	-20
i8	133-81	486	5	112	6	15	50	-1
apex6	135-99	438	6	189	12	-11	50	-58
x3	135-99	448	8	77	5	-2	-14	13
i6	138-67	267	2	56	4	3	60	-43
pair	173-137	841	10	166	12	7	17	-12
i4	192-6	74	3	464	15	0	40	-38
i7	199-67	327	2	112	5	-8	60	-46
i2	201-1	68	4	189	9	66	43	60
des	256-245	2398	11	189	8	-25	31	-55
i10	257-224	1515	21	189	8	17	40	19
Average		307	5.5	1083	7.9	7.4	41	-16

V. EXTENSION OF THE METHOD TO PARTIALLY DEFINED FUNCTIONS AND FOR MEASURING GENERALIZATION

We analyze here the ability of the algorithm to generalize in cases when only a subset of the instances of the Boolean function are known. The generalization ability is the capacity of predicting the correct output of a function for unseen inputs, i.e., instances of the functions that have not been used in the process of constructing the circuits. In general, within the area of circuit synthesis the whole Boolean function is specified and there is not much interest in the generalization ability of the methods. On the other hand, the issue about generalization is very much relevant in the areas of neural networks and pattern recognition where it has been extensively analyzed. The study of the VLSI implementation of neural circuits, an area in between the two previous ones is now attracting much attention and for this reason knowing the generalization ability of circuit synthesis tools has some relevance and has been also previously analyzed [30].

In order to analyze the generalization ability of DASG, the algorithm has to be modified so it can work with don't care symbols, ($*$), used for undetermined values. The use of don't care

symbols is necessary to represent the unknown instances of the target function and also it helps in the internal representation of the partial functions allowing for a more general circuits that will have a better generalization ability. The general scheme of the decomposition procedure, shown in Fig. 1 remains similar but the use of don't care symbols makes necessary some modifications in the function splitting procedure. The equations for the splitting procedure have to be modified and we give below the form of them, starting with the case when an OR representation is used. For this case of the OR representation, the splitting equations are such that the new function $F1_a$ takes the values of the original function F when the splitting variable x_i is equal to 0. As in the case of a fully defined function, the new $F1_a$ function takes null values when the splitting variable x_i is equal to 1, but in this case it is a condition that also happens that the value of F is 1. For values in which $x_1 = 1$ and the value of F is 0, the new function $F1_a$ can take an undetermined value, specified by the don't care symbol in the formulas. The value in these cases can go undetermined as these examples can take output values 0 or 1 without affecting the decomposition procedure, as the value of this example is taken by the function $F1_b$. In fact, these equations are also valid for the problem of the synthesis of architectures analyzed before and not only for generalization, but their application involves the use of don't care symbols. The notation $\vec{x}$ is used for the vector of inputs $(x_0, \ldots, x_i, \ldots, x_{n-1})$ in the following:

$$F1_a(\vec{x}) = \begin{cases} F(\vec{x}) & \text{if } x_i = 0 \\ 0 & \text{if } x_i = 1 \wedge F(\vec{x}) = 0 \\ * & \text{otherwise} \end{cases} \quad (3)$$

$$F1_b(\vec{x}) = \begin{cases} F(\vec{x}) & \text{if } x_i = 1 \\ 0 & \text{if } x_i = 0 \wedge F(\vec{x}) = 0 \\ * & \text{otherwise} \end{cases} . \quad (4)$$

For the case of using an AND representation, the equations have a similar form but 1's are used instead of 0's for the second condition in (3)–(4).

An example of the application of the generalization algorithm for the case of a partially defined function of 4 variables is illustrated in Fig. 4. On top of the figure, the partially defined function with truth vector $[10 * 110 * 1 * 1\ 1 * *1\ 0\ 1]$ is shown. The influence of the variables is computed for the nonunate variables and it is shown in the top box in a row indicated by S_{xi}. The function is nonunate and thus nonthreshold. The variable x_0 is the only nonunate variable and then is the one used for the splitting procedure, that creates two new functions according to (3)–(4). An elimination of the nonrelevant variables is then applied to reduce the dimensionality of the problem, followed by the linear separability tests of the functions. The procedure finishes because both created functions are threshold functions. The final architecture is shown near the bottom of Fig. 4, where the whole truth vector of the obtained function is also shown, including the prediction for the originally nondefined bits.

To test the generalization ability of the introduced algorithm, a set of 22 single output functions with a number of inputs between 5 and 14 were analyzed. A generalization ability around 100% was obtained for the rest of functions indicated in Table I using all four different methods considered even in the case of using 20% of the examples in the training set and thus they are not included in the comparison shown in Table III.

To compare the performance of the DASG algorithm for generalization, we used three alternatives standard approaches: the

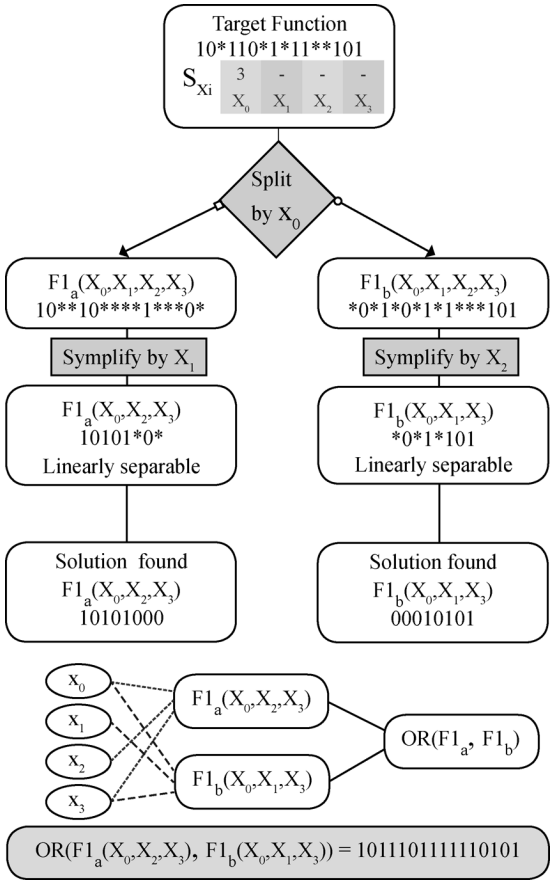

Fig. 4. An example of the application of the generalization algorithm for the case of a partially defined function. The algorithm check for the linear separability of the function, split it according to the variable with highest influence and simplify the function to eliminate irrelevant variables. In the example, the two newly created functions after the splitting is performed, are threshold functions. The solution, shown by the architecture on the bottom of the figure, computes a function that agrees with the defined function values and generalize the solution to the missing outputs.

C4.5 decision tree algorithm [39], feedforward neural networks (FFNN) trained by backpropagation and an implementation of the nearest neighboring algorithm for generalization (K-NN-gen). Many other classification algorithms exists but as it is not possible to make a comparison to all of them we selected three different widely used approaches.

In Table III, the results obtained from all four methods are shown, where it is possible to see that the best results where obtained by the new DASG algorithm with an average generalization ability of 89.78%, followed closely by FFNN (88.03%). The application of the C4.5 decision tree algorithm leads to the a generalization ability of 81.21% and similar results 82.38% of generalization were obtained with the K-NN-gen method. All the methods, except the introduced DASG algorithm, were applied using the open source platform WEKA using the standard parameter settings, as these parameters worked almost optimal in comparison to some other alternative tested sets. The K-NN-gen method was used instead of the standard K-nearest neighbors algorithm as the generalization ability obtained was larger. The difference between the generalization ability of the DASG and the C4.5 and K-NN-gen methods were statistically significant at $p = 0.009$ and $p = 0.012$, respectively, (paired

TABLE III
GENERALIZATION ABILITY OBTAINED WITH THE NEW INTRODUCED DASG ALGORITHM AND WITH THREE STANDARD ALGORITHMS, C4.5 DECISION TREES, FFNNAND NEAREST NEIGHBOR FOR GENERALIZATION K-NN-GEN ON SET OF 22 SINGLE OUTPUT FUNCTIONS (SEE TEXT FOR MORE DETAILS)

Function	# Inputs	Generalization ability			
		DASG	C4.5	FFNN	NN-Gen
cm82af	5	69.23	38.46	53.85	69.23
cm82ag	5	38.46	38.46	38.46	30.77
cm82ah	5	76.92	69.23	100.00	76.92
majority	5	61.54	69.23	61.54	61.54
z4ml24	7	96.08	76.47	96.08	82.35
z4ml25	7	96.08	70.59	100.00	39.22
z4ml26	7	100.00	56.86	100.00	86.27
z4ml27	7	100.00	60.78	100.00	100.00
9symml	9	91.71	82.93	100.00	79.51
alu2k	10	100.00	92.20	75.61	78.05
alu2l	10	81.71	82.20	68.78	73.66
alu2o	10	85.37	85.61	83.90	82.44
x2n*	10	98.05	98.29	98.53	98.29
x2p*	10	90.72	96.46	96.21	86.32
x2q*	10	96.58	84.98	97.56	90.23
cm85al	11	100.00	97.68	99.15	97.07
cm85am	11	99.51	97.31	98.29	96.58
cm85an	11	99.75	97.80	99.76	97.56
alu4q	14	99.57	98.32	86.15	95.76
alu4r	14	96.34	95.45	87.27	94.69
alu4u	14	97.68	97.76	95.44	96.67
cm162aq*	14	99.87	99.64	99.98	99.33
Average		89.78	81.21	88.03	82.38

t-test, d.f. $= 21$). For the comparison between DASG and FFNN the difference was not statistically significant ($p = 0.39$) as the difference in the average generalization ability obtained was much smaller.

In all but four cases, 60% of the total number of examples were used as training set and the remaining 40% was used to test the generalization ability. For the 4 functions, indicated with an $*$ in Table III, the split between training and generalization sets was 20%–80% because otherwise the generalization ability was 100% for all methods.

VI. DISCUSSION

A new algorithm for the threshold synthesis of Boolean functions has been introduced in this work. In essence, the method works directly with the truth table of a Boolean function decomposing it iteratively according to the variable with the highest influence. The algorithm includes a procedure for checking whether the new obtained functions are threshold functions, and for the non threshold ones the whole procedure is repeated iteratively until all functions found are linearly separable. When the algorithm is applied directly to the truth vector of the whole target function the algorithm creates architectures with a single hidden layer containing all the obtained functions that are combined at the output by an AND or OR output function. In practice, computational costs restrict, due to the exponential number of instances that compose the truth vector of a function, the direct application of the algorithm to functions with up to 21 inputs. However, by applying standard processing steps used in circuit design, the algorithm can be applied to functions with a very large number of inputs. To test the performance of the algorithm, we have applied it first directly to a subset of benchmark functions of up to 21 inputs and then we analyzed the performance by applying the algorithm in combination with some preprocessing performed using the SIS system using larger functions with a number of inputs of up to 257. The benchmark functions used are from the MCNC dataset and the results were compared to those recently obtained by Zhang *et al.* [25]. The results obtained from the direct application of the algorithm to the function truth table is presented in Table I and shows that the new algorithm is very efficient in terms of the number of gates and levels of the constructed architectures in comparison to the recent results from [25], as a reduction of 20.03% in the number of gates and 61.8% in the number of levels was obtained, with improvements of up to 85.1% in the number of gates for certain functions. The architectures constructed with the DASG algorithm have on average a larger number of connections than the ones constructed in [25] as an average increase of 29.0% was obtained for this dataset. When the DASG algorithm was applied in combination to some preprocessing using the SIS system to larger size functions, the improvements on the number of levels and number of gates of the architectures were a little bit reduced, but in any case significative, with an average of 7.4% reduction on the number of gates and 41% reduction on the number of levels in comparison to the results of Zhang *et al.*, 2005. For this second set of functions, the interconnect was also larger than the values reported by Zhang *et al.* with an increase of 16%.

One important aspect of the architectures generated by the DASG algorithm regards the obtained values of maximum fan-in used in the circuits and shown in the Tables I and II. The values of fan-in maximum for the obtained circuits have an average value of 13 for the first set of functions with a number of inputs up to 21, while a value of 7.9 was obtained for the second set of functions with larger input size. The method implemented by Zhang *et al.*, (2005) does not improve if the allowed fan-in max is increased beyond 6 or 7 due to the low probability of a function being threshold for high dimensions, and thus they present results with a fan-in limited to 6.

Furthermore, the generalization ability of the DASG algorithm was also tested on a set of 22 functions (shown in Table III) belonging to the MCNC benchmark dataset and the results were compared to the ones obtained from three different standard classification algorithms: C4.5 decision trees, standard feedforward neural networks and nearest neighbor classifiers. The generalization ability of the DASG algorithm was the largest of the four different method tested, achieving a value of 89.78%, FFNN came second with a generalization ability a little bit smaller 88.03% and a difference that was not statistically significant. For the other two methods, C4.5 decision trees and K-NN-gen, the generalization obtained was lower (81.21% and 82.38%, respectively).

As a final conclusion, a new algorithm for threshold synthesis and generalization of Boolean functions has been introduced and the results obtained so far show that the algorithm performs quite efficiently, both in terms of the size of the generated architectures and in the level of generalization ability obtained in comparison to existing approaches.

ACKNOWLEDGMENT

The authors acknowledge useful discussions with M. Anthony (LSE, London) and with J. M. Quintana and M. Avedillo (INM, Sevilla). They also acknowledge useful comments made by the reviewers.

REFERENCES

[1] R. O. Winder, "Threshold logic," Ph.D., Princeton Univ., Dep. Math., Princeton, NJ, 1962.

[2] M. L. Dertouzos, *Threshold Logic: A Synthesis Approach*. Cambridge, MA: MIT Press, 1965.

[3] J. E. Hopcroft and R. L. Mattson, "Synthesis of minimal threshold logic networks," *IEEE Trans. Electromagn. Compat.*, vol. EC-6, pp. 552–560, 1965.

[4] S. T. Hu, *Threshold Logic*. Berkeley, CA: Univ. California Press, 1965.

[5] S. Muroga, *Threshold Logic and its Applications*. New York: Wiley, 1971.

[6] I. Wegener, "The complexity of the parity function in unbounded fan-in, unbounded depth circuits," *Theoret. Comput. Sci.*, vol. 85, pp. 155–170, 1991.

[7] K. Y. Siu and V. P. Roychowdhury, "On optimal depth threshold circuits for multiplication and related problems," *SIAM J. Discrete Math.*, vol. 7, pp. 284–292, 1994.

[8] V. Bohossian, P. Hasler, and J. Bruck, "Programmable neural logic," *IEEE Trans. Compon., Packag., Manufact. Technol., Part B*, vol. 21, pp. 346–351, 1998.

[9] V. Beiu and J. G. Taylor, "On the circuit complexity of sigmoid feedforward neural networks," *Neural Netw.*, vol. 9, pp. 1155–1171, 1996.

[10] W. Maass, G. Schnitger, and E. D. Sontag, "A comparison of the computational power of sigmoid and Boolean threshold circuits," in *Theoretical Advances in Neural Computation and Learning*, V. P. Roychowdhury, K. Y. Siu, and A. Orlitsky, Eds. Boston, MA: Kluwer, 1994, pp. 127–151.

[11] G. B. Huang, Q. Y. Zhu, K. Z. Mao, C. K. Siew, P. Saratchandran, and N. Sundararajan, "Can threshold networks be trained directly?," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 53, pp. 187–191, 2006.

[12] M. Mezard and J. P. Nadal, "Learning in feedforward layered networks: The tiling algorithm," *J. Phys. A*, vol. 22, pp. 2191–2204, 1989.

[13] M. Frean, "The upstart algorithm: A method for constructing and training feedforward neural networks," *Neural Comput.*, vol. 2, pp. 198–209, 1990.

[14] D. L. Gray and A. N. Michel, "A training algorithm for binary feedforward neural networks," *IEEE Trans. Neural Netw.*, vol. 3, pp. 176–194, 1992.

[15] E. Mayoraz and F. Aviolat, "Constructive training methods for feedforward neural networks with binary weights," *Int. J. Neural Syst.*, vol. 7, pp. 149–166, 1996.

[16] S. A. J. Keibek, G. T. Barkema, H. M. A. Andree, M. H. F. Savenlie, and A. Taal, "A fast partitioning algorithm and a comparison of binary feedforward neural networks," *Europhys. Lett.*, vol. 18, pp. 555–559, 1992.

[17] H. M. A. Andree, G. T. Barkema, W. Lourens, A. Taal, and J. C. Vermeulen, "A comparison study of binary feedforward neural networks and digital circuits," *Neural Netw.*, vol. 6, pp. 785–790, 1993.

[18] K. Y. Siu, V. P. Roychowdhury, and T. Kailath, "Depth-size tradeoffs for neural computation," *IEEE Trans. Comput.*, vol. 40, pp. 1402–1412, 1991.

[19] A. L. Oliveira and A. Sangiovanni-Vincentelli, "LSAT: An algorithm for the synthesis of two level threshold gate networks," in *Proc. ACM/IEEE Int. Conf. Comput.-Aided Des.*, Santa Clara, CA, 1991, pp. 130–133.

[20] I. Parberry, *Circuit Complexity and Neural Networks*. Cambridge, MA: MIT Press, 1994.

[21] R. Impagliazzo, R. Paturi, and M. E. Saks, "Size-depth tradeoffs for threshold circuits," *SIAM J. Comput.*, vol. 26, pp. 693–707, 1997.

[22] W. Noth, U. Hinsberger, and R. Kolla, "TROY: A tree-based approach to logic synthesis and technology mapping," in *Proc. 6th Great Lakes Symp. VLSI*, 1996, pp. 188–193.

[23] V. Beiu, J. M. Quintana, and M. J. Avedillo, "LSI implementations of threshold logic – A comprehensive survey," *IEEE Trans. Neural Netw.*, vol. 14, pp. 1217–1243, 2003.

[24] M. J. Avedillo and J. M. Quintana, "A threshold logic synthesis tool for RTD circuits," in *Proc. Euromicro Symp. Digit. Syst. Design*, 2004, pp. 624–627.

[25] R. Zhang, P. Gupta, L. Zhong, and N. K. Jha, "Threshold network synthesis and optimization and its application to nanotechnologies," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 24, pp. 107–118, 2005.

[26] T. Gowda, S. Vrudhula, and G. Konjevod, "Combinational equivalence checking for threshold logic circuits," in *Proc. 17th Great Lakes Symp. VLSI*, 2007, pp. 102–107.

[27] P. Porwik and R. S. Stankovic, "Dedicated spectral method of Boolean function decomposition," *Int. J. Appl. Math. Comput. Sci.*, vol. 16, pp. 271–278, 2006.

[28] I. Gómez, L. Franco, J. L. Subirats, and J. M. Jerez, "Neural networks architecture selection: Size depends on function complexity," *Lecture Notes on Comput. Sci.*, vol. 4131, pp. 122–129, 2006.

[29] E. Sentovich, K. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. Stephan, R. Brayton, and A. Sangiovanni-Vincentelli, SIS: A System for Sequential Circuit Synthesis. Tech. Rep. Univ. California, Berkeley, 1992.

[30] M. Muselli and D. Liberati, "Training digital circuits with Hamming clustering," *IEEE Trans. Circuits Syst. I, Fundam. Theory Appl.*, vol. 47, pp. 513–527, 2000.

[31] E. Boros, P. L. Hammer, T. Ibaraki, A. Kogan, E. Mayoraz, and I. Muchnik, "An implementation of logical analysis of data," *IEEE Trans. Knowl. Data Eng.*, vol. 12, pp. 292–306, 1997.

[32] S. J. Hong, "R-MINI: An iterative approach for generating minimal rules from examples," *IEEE Trans. Knowl. Data Eng.*, vol. 9, pp. 709–717, 1997.

[33] H. A. Curtis, *A New Approach to the Design of Switching Circuits*. Princeton, NJ: Van Nostrand, 1962.

[34] R. Ashenhurst, "The decomposition of switching functions," in *Proc. Int. Symp. Theory of Switching Functions*, 1957, pp. 74–116.

[35] J. L. Subirats, I. Gómez, J. M. Jerez, and L. Franco, "Optimal synthesis of Boolean functions by threshold functions," *Lecture Notes on Comput. Sci.*, vol. 4131, pp. 983–992, 2006.

[36] L. Franco, J. L. Subirats, M. Anthony, and J. M. Jerez, "A new constructive approach for the set of linearly separable functions," in *Proc. IEEE Int. Joint Conf. Neural Netw.*, Vancouver, Canada, 2006, pp. 9541–9546.

[37] M. Berkelaar, The LP Solve Solver for Mixed Integer-Linear Programming 1997 [Online]. Available: http://www.cs.sunysb.edu/algorith/implement/lpsolve/implement.shtml

[38] L. Franco and S. A. Cannas, "Generalization properties of modular networks: Implementing the parity function," *IEEE Trans. Neural Netw.*, vol. 12, pp. 1306–1313, 2001.

[39] J. R. Quinlan, *C4.5: Programs for Machine Learning*. San Mateo, CA: Morgan Kauffman, 1992.

[40] L. Franco, "Generalization ability of Boolean functions implemented in feedforward neural networks," *Neurocomputing*, vol. 70, pp. 351–361, 2006.

[41] L. Franco and M. Anthony, "The influence of oppositely classified examples on the generalization complexity of Boolean functions," *IEEE Trans. Neural Netw.*, vol. 17, pp. 578–590, 2006.

[42] S. Muroga, "The principle of majority decision elements and the complexity of their circuits," in *Proc. Int. Conf. Inf. Process.*, 1959, pp. 400–407.

[43] I. Wegener, *The Complexity of Boolean Functions*. New York: Wiley, 1987.

José L. Subirats received the M.S. degree in computer science from the University of Málaga, Málaga, Spain, in 2007. He is currently working toward the Ph.D. degree at the University of Málaga.

His research interests include constructive algorithms for neural networks and genetic algorithms.

José M. Jerez received the Ph.D. degree in computer science in 2003 from the University of Málaga, Málaga, Spain.

He is currently an Associate Professor with the University of Málaga. His research interests lie in the areas of computational intelligence, image analysis, and bioinformatics. In particular, he is developing prediction software for biomedical problems using artificial intelligence techniques in collaboration with the Málaga University hospital. He has participated in more than 15 research projects funded by international and national boards and he belongs to several reviewing scientific committees.

Leonardo Franco (M'06) received the M.S. and Ph.D. degrees in physics from the University of Córdoba, Argentina, analyzing the generalization properties of feedforward neural networks.

He later was a Postdoctoral Fellow to SISSA, Trieste, Italy, where he became involved in Computational Neuroscience. He then joined the University of Oxford, U.K., where he applied and developed information theory methods to the analysis of neuronal recordings. He is presently with Málaga University, Málaga, Spain, as a Ramón y Cajal researcher working on neural networks, their applications to biomedical problems and also in computational neuroscience. He has authored approximately 35 publications in journals and international conferences.