A Simple and Effective Heuristic Method for Threshold Logic Identification

Augusto Neutzling, *Student Member, IEEE*, Mayler G. A. Martins, *Member, IEEE*, Vinicius Callegaro, *Member, IEEE*, André I. Reis, *Senior Member, IEEE*, and Renato P. Ribas, *Member, IEEE*

***Abstract*—In this paper, a straightforward and effective method to identify threshold logic function (TLF) is presented. Threshold logic is a promising alternative to conventional Boolean logic due to its suitability for emerging technologies, like memristors, quantum-dot cellular automata, resonant tunneling device, and spintronic devices. Identification and synthesis of TLF are essential tasks in a design flow based on such a logic strategy. The proposed method relies on irredundant sum-of-products Boolean function description form and exploits both ordering of variables and system of inequalities to assign the variable weights and the function threshold value. This is the first heuristic algorithm able to identify all threshold functions with up to six variables, being also more effective than other heuristic methods for functions with a larger number of variables. For functions obtained from *k*-cuts of benchmark circuits, experimental results demonstrated effectiveness near to 100% when compared to exact methods, even when the number of function variables increases. The execution time of the proposed approach is similar to related heuristic methods, being faster than integer linear programming-based algorithms.**

***Index Terms*—Digital circuit, linear separable function, threshold functions, threshold logic gate, threshold logic identification.**

I. Introduction

The limits of MOS transistor shrinking have made necessary the investigation of new alternatives to very large-scale integration circuit design. Among potential candidates are the most recent nanodevices like memristors, resonant tunneling devices, quantum-dot cellular automata, single electron transistors, and spintronic devices [1]–[3]. In this context, the proposed strategies to design digital integrated circuits based on such new technologies seem to be more effective through the implementation of threshold logic functions (TLFs) into single gates, i.e., by building threshold logic gates (TLGs). For this reason, threshold logic synthesis has been recently revisited aiming to explore the particularities of this kind of Boolean functions [2]–[5]. Moreover, the construction of TLGs has been proposed for both CMOS and emerging nanometric technologies [5]–[8], [16].

In this scenario, the threshold logic identification is an essential task that corresponds to the process of identifying whether a given Boolean function is TLF, besides computing the variable weights and function threshold value. Notice that the most of the identification methods available in the literature are based on solving systems of inequalities generated from truth table description form. These methods exploit integer linear programming (ILP) to provide optimal results [2], [4], [9]. However, scalability is their main bottleneck because the system of inequalities tends to increase exponentially with the number of function variables. One of the first heuristic (non-ILP-based) methods to identify threshold functions was proposed by Gowda *et al.* [10], and later improved in [11]. Gowda *et al.*'s [10] method applies functional decomposition and min-max factorization tree techniques. The target function is decomposed into simple subfunctions until these ones can be directly identified as AND and OR basic functions, or even as constant logic values “0” and “1.” These subfunctions are then merged by exploiting TLF properties. Palaniswamy *et al.* [12] proposed a method-based directly on the Chow's parameters, being later improved in [13]. However, the main drawbacks of Palaniswamy *et al.*'s [12] approach are the degradation on the number of identified TLFs and the fact that the assigned variable weights do not always correspond to the minimum possible values. These nonminimum weights may impact the final circuit area [2], [5].

This paper presents a novel non-ILP-based method to perform the identification and synthesis of TLF. This new algorithm is inspired by previous work presented in [14], being based on the manipulation of inequalities generated from irredundant sum-of-products (ISOPs) function representation in order to assign the TLF variable weights. The proposed approach presents three major contributions.

1) A new heuristic method to assign the variable weight values of TLFs.
2) A novel ISOP-based procedure able to define the variable weight ordering (VWO) before computing the absolute variable weight values.
3) A fast algorithm to generate ISOP representation of unate Boolean functions.

To the best of our knowledge, the proposed method is the first heuristic approach capable of identifying all TLFs with up to six variables. The method can be applied in

Manuscript received August 30, 2016; revised January 15, 2017 and April 6, 2017; accepted May 26, 2017. Date of publication July 19, 2017; date of current version April 19, 2018. This work was supported by the Brazilian funding agencies CAPES, CNPq, and FAPERGS. This paper was recommended by Associate Editor S.-C. Chang. *(Corresponding author: Augusto Neutzling.)*

A. Neutzling, A. I. Reis, and R. P. Ribas are with the Institute of Informatics, Federal University of Rio Grande do Sul, Porto Alegre 90040-060, Brazil (e-mail: ansilva@inf.ufrgs.br; andreis@inf.ufrgs.br; rpribas@inf.ufrgs.br).

M. G. A. Martins is with Carnegie Mellon University, Pittsburgh, PA 15213 USA (e-mail: mayler@cmu.edu).

V. Callegaro is with Mentor-A Siemens Business, Fremont, CA 94538 USA (e-mail: vinicius_callegaro@mentor.com).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCAD.2017.2729403

threshold logic synthesis tools finding TLF with more variables without loss of efficiency [2], [4], [11], [15], [16], [27]. Moreover, the proposed method is also able to identify more TLFs with a larger number of variables in comparison to other non-ILP-based methods, maintaining similar execution time. For functions obtained from k-cuts of Altera opencores circuits [25], experimental results have demonstrated that the effectiveness is near to 100% when compared to exact methods, even when the number of inputs increases.

The rest of this paper is organized as follows. In Section II, we present some preliminaries about Boolean functions and threshold logic. The proposed method is described in Section III. In Section IV, two case studies are presented in order to demonstrate the application of the method. Section V presents experimental results, proving the algorithm efficiency through an extensive comparison to the state-of-the-art approaches. The conclusions are outlined in Section VI.

II. Preliminaries

Some definitions and fundamentals are presented for a better comprehension of the proposed method.

A *Boolean function* f defined over the variable set $X = \{x_1, \dots, x_n\}$ is a function defined as $f(X) : B^n \to B$, where $B = \{0, 1\}$ and $n = |X|$, i.e., n is the number of variables in X. In this paper, AND, OR, and NOT operations are denoted by "$\cdot$," "$\vee$," and "!," respectively.

A *literal* is a variable (x_i) or its complement ($!x_i$), whereas a *cube* is a product of literals that represents a Boolean subspace. The *cube size* of a cube with l literals in a Boolean space B^n is given by $2^{(n-l)}$. Consequently, the size of a cube is inversely proportional to the number of literals in this cube.

Furthermore, an expression is called sum-of-products (SOPs) when this expression corresponds to product terms (AND) joined by a sum (OR) operation. In particular, an ISOPs is an SOP in which neither a literal nor a cube can be removed without changing the function behavior.

Given a function f represented by an ISOP form, this function is *unate* if, and only if, either direct (positive polarity) or complemented (negative polarity) literals, but not both, appear to each variable. Notice that a unate function has a unique ISOP representation [17].

By considering a set of all functions with up to n variables, these functions can be grouped into *classes*. Boolean functions can be grouped taking into account the negation (N), and/or the permutation (P) of variables, and/or the negation of function value [18]. For instance, NP-class corresponds to the set of distinct functions obtained by negating and/or permuting the input variables.

On the other hand, in the threshold logic context, TLF is a Boolean function that fits in the following structure. Each variable has a corresponding weight and the Boolean function has a threshold value. If the sum of weights of variables presenting true (1) value is greater than or equal to the threshold value, then the function evaluates to true (1) value. Otherwise, the function is false (0). This behavior can be expressed as follows [19]:

$$f = \begin{cases} 1, & \sum_{i=1}^{n} w_i \cdot x_i \geq T \quad \text{(1a)} \\ 0, & \text{otherwise} \quad \text{(1b)} \end{cases}$$

$$\begin{tikzcd}
\begin{array}{c} f(x) \\ {[w;\ T]} \end{array} \arrow[r, "!f", "(1)"'] \arrow[d, "!x"'] & \begin{array}{c} !f(x) \\ {[-w;\ 1\text{-}T].} \end{array} \arrow[d, "!f(!x)"] \\
\begin{array}{c} {[-w;\ T\text{-}\sum w_i]} \\ f(!x) \end{array} \arrow[r, "!f"] & \begin{array}{c} {[w;\ 1\text{-}T+\sum w_i]} \\ f^{d}(x) \end{array}
\end{tikzcd}$$

Fig. 1. Elementary properties of TLFs [19].

where x_i represents each variable [0, 1], w_i is the corresponding weight of each variable, and T is the threshold value of the function. A threshold function can be completely represented by the vector $[w_1, w_2, \dots, w_n; T]$. Notice that NAND and NOR Boolean functions are TLF, as well as some complex functions as the following one:

$$f_1 = (x_1 \cdot x_2) \vee (x_1 \cdot x_3) \vee (x_2 \cdot x_3 \cdot x_4) \vee (x_2 \cdot x_3 \cdot x_5) \quad (2)$$

that corresponds to the TLF represented by $f = [4, 3, 3, 1, 1; 7]$.

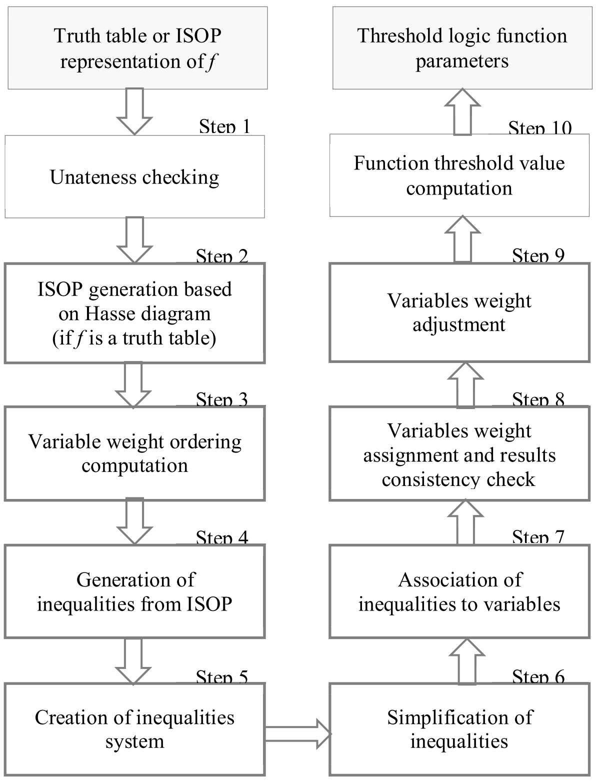
All TLFs are unate functions but not all unate functions are TLF. Therefore, if a function has binate variables then it is not TLF [19]. For instance, the function $f_2 = (x_1 \cdot x_2) \vee (x_3 \cdot x_4)$ is a unate function that is not TLF. Notice that, when a given unate Boolean function presents variables in negative polarity, these variables can be treated in the same way as functions having all variables in positive polarity in order to identify whether it is TLF. If $f(x_1, x_2, \dots, x_n)$ is TLF, represented by $[w_1, w_2, \dots, w_n; T]$, then the complemented function $!f(x_1, x_2, \dots, x_n)$ is also TLF, defined by $[-w_1, -w_2, \dots, -w_n; (1 - T)]$. This property is illustrated in Fig. 1.

In terms of circuit design, a TLG corresponds to the physical construction of a given TLF into a single gate. As discussed above, some complex Boolean functions can be identified as TLF. Thus, respecting the restrictions and limitations of the target technology, a promising feature of threshold logic design is the reduction in the number of gates instantiated in the circuit, and probably the associated physical area [2], [5].

Several works have proposed TLGs built on different technologies, based both on conventional static CMOS design and emerging technologies [5]–[8], [16]. These works also analyze electrical characteristics of gates.

In addition to signal delay propagation, circuit area and power dissipation evaluation, the reliability of TLG is also an important concern. In particular, using minimum weights can decrease the reliability of the TLG. In this sense, some works have exploited a tradeoff between area and reliability where the variable weights are increased in order to improve the circuit reliability [29], [30]. These methods consider that the variability on the manufacturing process and the signal noise can be modeled as perturbations in the variable weights and function threshold value.

The identification method proposed herein tries to compute minimum input weights, aiming the area optimization. Notice that obtaining minimum weights is the first action in a reliability aware flow. Therefore, in a future work, the proposed method can be extended by adding a reliability factor for each input weight assignment.



Truth table or ISOP representation of f
Step 1
Unateness checking
Step 2
ISOP generation based on Hasse diagram (if f is a truth table)
Step 3
Variable weight ordering computation
Step 4
Generation of inequalities from ISOP
Step 5
Creation of inequalities system
Simplification of inequalities
Step 6
Association of inequalities to variables
Step 7
Variables weight assignment and results consistency check
Step 8
Variables weight adjustment
Step 9
Function threshold value computation
Step 10
Threshold logic function parameters

Fig. 2. Flowchart of the proposed algorithm for TLF identification.

III. Threshold Logic Function Identification

The most common methodology adopted to identify TLF performs the generation of a system of inequalities from the truth table representation and then solves this system by using ILP approach [2], [4], [9]. If a solution for such inequalities system is found, the given function is considered TLF and the solution corresponds to the variable weights and the threshold value. In the case that the system cannot be solved, the function is not TLF and more than one TLF is required to implement the target function.

In our method, a complete system of inequalities is also built using a similar strategy to ILP inequalities generation algorithms. However, unlike ILP-based approaches, the inequalities system is not solved. Instead, the algorithm selects some of the inequalities as constraints to the associated variables to compute the variable weights in a bottom-up way. After this assignment, the consistency of the complete system is verified in order to check whether the weights have been correctly computed.

For the sake of comprehension, the algorithm has been split into ten steps, illustrated in Fig. 2, as summarized in the following. Step 1 checks the unateness of the given Boolean function; if the function is unate, then the ISOP is generated in step 2, except whether the ISOP has already been applied as input function description; in step 3, the ordering of the variable weights is identified; step 4 generates the inequalities; step 5 creates the system of inequalities; in step 6, the simplification of the inequalities set is executed; in step 7, the association of each variable to some inequalities is performed; and step 8 assigns the variable weights while the consistency of the solution is verified and whether the given Boolean function is confirmed as TLF. In step 9, it is performed an eventual adjustment of the weights, when necessary. Finally, the threshold value is computed in step 10.

Algorithm 1 Function Unateness Checking

Input: Function f *in truth table form*
Output: Vector U containing the unateness of each variable

```
1:  for each $x_i \in X$
2:      $\alpha = f(x_i = 1)$
3:      $\beta = f(x_i = 0)$
4:      $\gamma = \alpha \vee \beta$
5:      if ($\alpha = \beta$) then $U\ [x_i] = don't\_care$
6:      else if ($\alpha \equiv \gamma$) then $U\ [x_i] = positive\_unate$
7:      else if ($\beta \equiv \gamma$) then $U\ [x_i] = negative\_unate$
8:      else if ($\alpha \neq \beta \neq \gamma$) then
9:          $U\ [x_i] = binate$
10:         return FALSE
11:     end if
12: end for
13: return $U$
```

A. Unateness Checking

As mentioned before, if a given function is not unate then it is not TLF. Therefore, in step 1, the algorithm checks the function unateness property. For binate function, this step already returns FALSE, i.e., the function is not TLF.

A completely specified function is positive unate in variable x, iff$(x_i = 1) \supset f(x_i = 0)$, where $f(x_i = 1)$ is the positive cofactor and $f(x_i = 0)$ is the negative cofactor of f with respect to variable x_i. The unateness checking is based on the positive and negative cofactors generation. Algorithm 1 presents the pseudocode of the unateness checking, considering the input as a truth table that corresponds to step 1 of the method, depicted in Fig. 2. If the input is an ISOP form, the unateness checking is trivially performed by iterating on all cubes and storing the polarities of each variable. If both polarities of at least one variable appear, the function is not unate.

As discussed in Section II, considering threshold functions, a negative unate variable can be changed to a positive one by just inverting the weight signal, and this amount is then subtracted from the threshold value. In our method, the negative variables are treated as positive ones, and this information is stored. After computing the threshold function parameters, the input weights are adjusted.

For instance, in the given function $f_{\text{original}} = (!a \vee (b \cdot c))$, the variable a is negative unate. The method considers the function $f_{\text{positive}} = (a \vee (b \cdot c))$, where a is positive unate. The identified variable weights for f_{positive} are 2, 1, and 1. Afterwards, the signal of the negative variables and the function threshold value are adjusted based on the properties illustrated in Fig. 1. The variable weights for f_{original} are -2, 1, and 1, respectively.

B. ISOP Generation Based on Hasse Diagram

The method proposed herein for threshold logic identification works over an ISOP representation of a given Boolean function f and an ISOP representation of $!f$. The ISOP needs to be generated when either the input function is represented by a truth table (usually for functions with up to 16 variables) or an ISOP of $!f$ is not provided.

The conventional methods adopted in the ISOP generation usually apply the Espresso logic minimizer [17] and the Minato [20] algorithm, being suitable for general (unate and

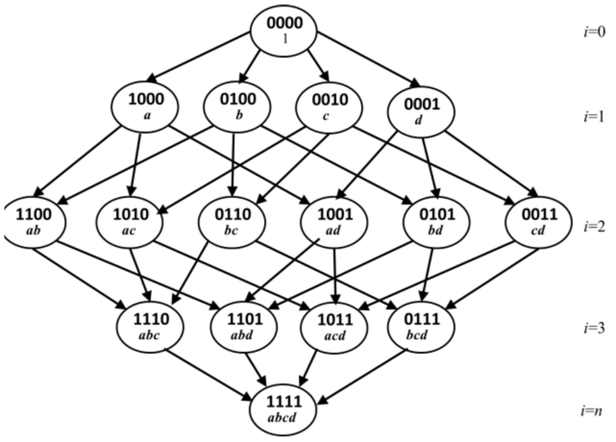


Fig. 3. Hasse diagram of a four-input Boolean function [21].

Algorithm 2 Compute the ISOP of a Positive Unate Function Based on the Hasse Diagram

Input: Boolean function f
Output: Set of cubes *isop*

```
1:  queue = Ø
2:  isop = Ø
3   status[ ] = Ø
4:  $f_p$ = Ø
5:  for each $x_i \in X$
6:      queue.add ($2^i$)
7:  while(queue is not empty)
8:          cube = queue.remove
9:          if (status[parentrof(cube)] = ACCEPTED)
10:                 continue
11:         $f_c$ = getfunction(cube)
12:         if ($f = f \vee f_c$)
13:                 isop = isop ∪ cube
14:                 status[cube] = ACCEPTED
15:                 $f_p = f_p \vee f_c$
16:         else
17:                 queue.add (childrenof(cube))
18:         if($f_p = f$)
19                  return isop
20: end while
21: return isop
```

binate) Boolean functions. In this paper, we developed a faster procedure created specifically to handle unate functions.

This novel ISOP generation algorithm is based on the Hasse diagram, where each vertex corresponds to a minterm, each level i contains cubes with i literals, and each cube covers all of its descendants in the diagram [21]. A breadth-first search is performed, such that larger cubes, i.e., cubes that cover more minterms, are visited before the smaller cubes. For instance, a complete Hasse diagram of a four-input Boolean function is illustrated in Fig. 3.

For each visited vertex in the diagram, if the respective cube function f_c is contained in the target function f, then this cube is added to the partial solution f_p and its descendants are not visited. When the partial solution function is equivalent to the target function ($f \equiv f_p$), the final solution is found. The ISOP generation procedure is described by the pseudocode in Algorithm 2 and corresponds to step 2 of the flowchart depicted in Fig. 2.

Algorithm 3 Compute the VWO

Input: Function f with n variables represented by an ISOP F with cube set C that contains m cubes
Output: List VWO parameters *list_vwo* in ascending order

```
1: initialize all values of list_vwo as zero
2: for each cube $c \in C$ do
3:     $lit = |c|$
4:     for each $x_i \in c$ do
5:         add $m^{(n-lit)}$ in list_vwo[$x_i$]
6:     end for
7: end for
8: order(list_vwo)
9: return list_vwo
```

C. Variable Weight Ordering Computation

In the proposed method, the computation of the VWO it is a crucial task because this information is used in the inequalities simplification and the weight assignment steps. A well-known way to obtain such an ordering is through the Chow's parameters [17], [19]. The correlation between the Chow's parameters p_i and p_j of two variables x_i and x_j, respectively, induces the correlation between the corresponding weights w_i and w_j, i.e., if $p_i > p_j$ then $w_i > w_j$ [22].

A new algorithm to obtain a VWO parameter is presented in this paper. VWO parameters provide similar VWO as the Chow's ones, although the absolute parameter values are possibly different. These parameters are calculated directly from the ISOP, being quite straightforward and fast to compute. They are based on the *max literal* computation, as proposed in [11].

Considering an ISOP representation of a given function f, the largest variable weight of the target TLF is associated to the literal that occurs most frequently in the largest cubes of f, i.e., in the cubes with fewer literals. In the case of a tie, it is decided by comparing the frequency of the literals in the next cubes with the smaller size.

The algorithm defines a weight for each cube, corresponding to the cube size. This weight is added to the VWO parameter of the variables present in the cube. The VWO is associated with the ordering of these parameters computed for each variable. The pseudocode of step 3 is described in Algorithm 3.

In [14], the variable ordering is obtained through the Chow's parameters computation, whose time complexity is always 2^n for each variable. The complexity to compute the VWO parameter of each variable depends on the number of ISOP cubes, which is strictly smaller than 2^n. For unate functions, the worst case of the number of cubes is $(n!/\lfloor(n/2)\rfloor! \cdot \lceil(n/2)\rceil!)$.

For instance, for a given Boolean function defined by the following ISOP:

$$f_3 = (x_1 \cdot x_2) \vee (x_1 \cdot x_3 \cdot x_4) \quad (3)$$

the calculated values of variables x_1, x_2, x_3, and x_4 are $6 = 2^2 + 2^1, 4 = 2^2, 2 = 2^1$, and $2 = 2^1$, respectively, whereas the Chow's parameters for these variables are 10, 6, 2, and 2, respectively. Notice that the same ordering is obtained in both calculations. Thus, in this case, the algorithm assigns initially the weight of the variables x_3 and x_4, and then the weight of the variable x_2, being the weight of variable x_1 the last one to be assigned.

TABLE I
INEQUALITIES FROM TRUTH TABLE REPRESENTING THE FUNCTION IN (3)

x_1	x_2	x_3	x_4	f_3	Inequality
0	0	0	0	**0**	$0 < T$
0	0	0	1	**0**	$w_4 < T$
0	0	1	0	**0**	$w_3 < T$
0	0	1	1	**0**	$(w_3 + w_4) < T$
0	1	0	0	**0**	$w_2 < T$
0	1	0	1	**0**	$(w_2 + w_4) < T$
0	1	1	0	**0**	$(w_2 + w_3) < T$
0	1	1	1	**0**	$(w_2 + w_3 + w_4) < T$
1	0	0	0	**0**	$w_1 < T$
1	0	0	1	**0**	$(w_1 + w_4) < T$
1	0	1	0	**0**	$(w_1 + w_3) < T$
1	0	1	1	**1**	$(w_1 + w_3 + w_4) \geq T$
1	1	0	0	**1**	$(w_1 + w_2) \geq T$
1	1	0	1	**1**	$(w_1 + w_2 + w_4) \geq T$
1	1	1	0	**1**	$(w_1 + w_2 + w_3) \geq T$
1	1	1	1	**1**	$(w_1 + w_2 + w_3 + w_4) \geq T$

D. Generation of Inequalities From ISOP

Equation (1) defines the relationship between the variable weights and the threshold value of the target TLF. If the function value is true (1) for certain assignment vector then the sum of weights of this assignment is equal to or greater than the threshold value. Otherwise, the function value is false (0), i.e., the sum of weights is less than the threshold value. From these relationships, it is possible to generate the associated inequalities. As illustration, the relationships between the variable weights and the threshold value in respect with the truth table of the function from (3) are shown in Table I.

Some of these relationships are redundant because some inequalities are self-contained into other ones. For instance, once we have the relationship $(w_1 + w_2) \geq T$ and the variable weights are always positive, so the relationship $(w_1 + w_2 + w_3) \geq T$ is redundant. The irredundant information is the lesser assignments (i.e., the lesser weight sum) that make the function true (1) and the greater assignments (i.e., the greater weight sum) that make it false (0). Notice that an assignment vector $A(a_1, a_2, a_3 \ldots a_n)$ is smaller than or equal to an assignment vector $B(b_1, b_2, b_3 \ldots b_n)$, denoted as $A \leq B$, if, and only if, $a_i \leq b_i$ for $(i = 1, 2, 3, \ldots, n)$. For instance, the assignment vector (1,0,0,1) is lesser than the assignment vector (1,1,0,1), whereas the assignment vectors (0,1,0,1) and (1,1,0,0) are not comparable.

In our method, these redundancies are avoided using two ISOP expressions, one for the direct function f and another for the negated function $!f$. In the example above, the least true assignment vectors are (1,1,0,0) and (1,0,1,1), and the greatest false assignment vectors are (1,0,1,0), (1,0,0,1), and (0,1,1,1). Therefore, the algorithm creates only $(w_1 + w_2)$ and $(w_1 + w_3 + w_4)$ on the *greater side*, and $(w_1 + w_3)$, $(w_1 + w_4)$, and $(w_2 + w_3 + w_4)$ on the *lesser side*. The ISOP expressions of f, and $!f$ are considered as inputs to the proposed method. Each sum of variable weights greater than the function threshold value is placed on the *greater side* set, whereas each sum of weights which is less than the threshold value belongs to the *lesser side* set. Table II shows these two sets for function f_3, defined in (3).

TABLE II
INEQUALITIES SYSTEM GENERATED FOR FUNCTION DESCRIBED IN TABLE I

greater side				*lesser side*
(w_1+w_2)	$\geq$	T	$>$	(w_1+w_4)
$(w_1+w_3+w_4)$	$\geq$	T	$>$	(w_1+w_3)
---		T	$>$	$(w_2+w_3+w_4)$

Algorithm 4 Generation of Inequalities Sides

Input: ISOP form of function f and negated function $!f$
Output: Two sets of inequalities sides, a set *ineq_greater* and a set *ineq_lesser*

```
1: for each cube $c \in C$
2:     create inequality_side $S$ from $c$
3:     add $S$ in ineq_greater
4: end for
5: for each cube $c \in C'$
6:     create inequality_side $S$ from $c$
7:     add $S$ in ineq_lesser
8: end for
9: return <ineq_greater, ineq_lesser>
```

TABLE III
INEQUALITIES SYSTEM GENERATED TO THE FUNCTION DESCRIBED IN TABLE I

#	Inequality		
1	(w_1+w_2)	$>$	(w_1+w_4)
2	(w_1+w_2)	$>$	(w_1+w_3)
3	(w_1+w_2)	$>$	$(w_2+w_3+w_4)$
4	$(w_1+w_3+w_4)$	$>$	(w_1+w_4)
5	$(w_1+w_3+w_4)$	$>$	(w_1+w_3)
6	$(w_1+w_3+w_4)$	$>$	$(w_2+w_3+w_4)$

This procedure corresponds to step 4 of the flowchart depicted in Fig. 2, and is described by the pseudocode in Algorithm 4. The time complexity of this algorithm is $O(m + m')$, where m is the number of cubes in the ISOP of function f and m' is the number of cubes in the ISOP of the negated function $!f$.

E. Creation of Inequalities System

The pseudocode that represents the procedure to create the system of inequalities, corresponding to step 5 illustrated in Fig. 2, is presented in Algorithm 5. The instruction *compose_inequality* creates a new inequality from the two inequality sides. Each element on the *greater side* is greater than each element on the *lesser side*, and the *greater side* elements are greater than (or equal to) the threshold value, whereas the *lesser side* elements are smaller than that. The inequalities system is generated by performing a Cartesian product of the *greater side* set and the *lesser side* set.

Table III shows the six inequalities generated by the Boolean function illustrated in Table I. Notice that if the procedure had taken into account all the truth table assignments then 55 inequalities had been generated. The time complexity of Algorithm 5 is $O(m \cdot m')$, where m is the number of cubes in the ISOP of function f and m' is the number of cubes in the ISOP of negated function $!f$.

Algorithm 5 Inequalities System Generation

Input: Two sets of inequalities: a set *ineq_greater* and a set *ineq_lesser*
Output: Set of inequalities *ineq_set*

```
1: set_ineq = ∅
2: for each inequality_side $g \in$ ineq_greater
3:     for each inequality_side $l \in$ ineq_lesser
4:         ineq = compose_inequality($g$,$l$)
5:         add ineq in set_ineq
6:     end for
7: end for
8: return <set_ineq>
```

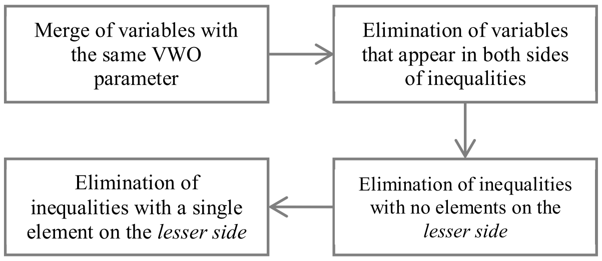

Merge of variables with the same VWO parameter
Elimination of variables that appear in both sides of inequalities
Elimination of inequalities with no elements on the lesser side
Elimination of inequalities with a single element on the lesser side

Fig. 4. Sequential tasks for inequalities simplification.

TABLE IV
UPDATED VARIABLES BASED ON VWO PARAMETERS FROM (3)

Input weight	VWO parameter	Updated variable
w_1	10	A
w_2	6	B
w_3 and w_4	2	C

F. Simplification of Inequalities

Besides generating only irredundant inequalities, the proposed method also simplifies each inequality and eventually discards some of them. The inequalities simplification process is performed through four basic tasks, as shown in Fig. 4.

The method assumes that if two variables have similar VWO parameters value then they present the same weight (it was proved by Muroga *et al.* [23] for functions with up to seven variables). Based on this assumption, the algorithm creates a new reduced set of variables where each variable corresponds to a VWO parameter value. These variables are called *updated variables* and are represented by A, B, C, and others, where A corresponds to the variable with the greatest VWO parameter value, B is the next, and so on.

The reduction in the number of variables allows the reduction of inequalities and, consequently, decreases the algorithm runtime. Table IV shows the updated variable created to the function described in (3), where the new number of variables is now three instead of four. Table V presents the new inequalities system with the updated variables. This domain transformation is described in Algorithm 6, whose time complexity is $O(n^2)$, being n the original number of variables.

The inequality simplification occurs when the variable weight appears on both sides of a given inequality. When it occurs, this variable is removed from the inequality. For instance, consider the inequality $(A+C+C)>(B+C+C)$. This inequality can be simplified by removing C, so resulting in the inequality $A > B$. This procedure is illustrated in Table VI. The resulting set of inequalities is presented in Table VII.

TABLE V
INEQUALITIES FROM TABLE III REPRESENTED BY NEW UPDATED VARIABLES

#	Inequality		
1	A+B	>	A+C
2	A+B	>	A+C
3	A+B	>	B+C+C
4	A+C+C	>	A+C
5	A+C+C	>	A+C
6	A+C+C	>	B+C+C

Algorithm 6 Domain Transformation for Inequalities Simplification

Input: Set of variables *X*, VWO parameters *V*, and inequalities *set_ineq*
Output: Set of inequalities *set_ineq'* with updated variables, grouped by VWO values

```
1: set_ineq' := ∅
2: set_tuples := ∅
3: for each VWO $v_i \in V$
4:     $v_i$_set := ∅
5:     for each variable $x_i \in X$
6:         if (vwo($x_i$) = $v_i$)
7:             add $x_i$ in $v_i$_set
8:         end if
9:     end for
10:    $x_j$' := get_first_element($v_i$_set)
11:    create tuple $t$ < $v_i$_set, $x_j$' >
12:    add $t$ in set_tuples
13: end for
14: set_ineq' := set_ineq
15: for each inequality ineq ∈ set_ineq'
16:     change_variables (ineq,set_tuples)
17: end for
18: return set_ineq'
```

TABLE VI
SIMPLIFICATION OF INEQUALITIES FROM TABLE V

#	Inequality		
1	~~A~~+B	>	~~A~~+C
2	~~A~~+B	>	~~A~~+C
3	A+~~B~~	>	~~B~~+C+C
4	~~A~~+~~C~~+C	>	~~A~~+~~C~~
5	~~A~~+~~C~~+C	>	~~A~~+~~C~~
6	A+~~C~~+~~C~~	>	B+~~C~~+~~C~~

TABLE VII
RESULTING SET OF INEQUALITIES FROM TABLE VI

#	Inequality		
~~1~~	B	>	C
~~2~~	B	>	C
3	A	>	C+C
~~4~~	C	>	0
~~5~~	C	>	0
~~6~~	A	>	B

Since all variable weights are positive, the algorithm discards the inequalities that have null weight (i.e., a weight equal to zero) on the *lesser side*. These inequalities are not useful because they only confirm that the weights are positive.

Algorithm 7 Inequalities Simplification

Input: Set of inequalities *set_ineq*
Output: Simplified set of inequalities *set_ineq_simplified*

```
1: set_ineq_simplified := Ø
2: for each inequality ineq ∈ set_ineq
3:     split ineq in greater and lesser side
4:     for each variablev1 ∈ greater_side
5:         for each variablev2 ∈ lesser_side
6:             if ($v1 = v2$)
7:                 remove(v1, greater_side)
8:                 remove(v2, lesser_side)
9:             end if
10:        end for
11:    end for
12:    if lesser_side is not empty
13:        ineq := compose_inequality (greater,lesser)
14:        add ineq in set_ineq_simplified
15:    end if
16: end for
17: return set_ineq_simplified
```

The inequalities which contain only one element on each side are checked just once using directly the VWO parameters. If one of these inequalities is not consistent, the function is defined as not TLF because the VWO parameter ordering is not respected. For instance, for inequality #1 in Table VII ($B > C$), the variables are replaced by the VWO parameter values, obtaining $6 > 2$. In this case, the VWO order holds and then the inequality is discarded.

Notice that, until this moment:

1) the method has not assigned the input weights yet;
2) the method is not able to determine whether a given Boolean function is TLF yet;
3) it is only possible to determine some functions that have been identified as non-TLF.

After all these simplifications, the set of useful inequalities can be significantly reduced. In the demonstration example, described in Table I, inequality #3 is the only one remaining in Table VII, that is $A > (C + C)$. As a result, it is the only inequality to be used in the assignment step. The reduction in the number of inequalities is a key factor of the proposed method since the weight assignment is based on inequalities manipulation. However, such a simplification makes the method heuristic, i.e., discarded inequalities could be essential for the right solution. It can generate false negatives for functions with more than six variables. The pseudocode of the complete inequalities simplification process, corresponding to step 6 illustrated in Fig. 2, is shown in Algorithm 7. The time complexity of this step is $O(n \cdot \log(n) \cdot m \cdot m')$, being n the number of variables, m the number of cubes in the ISOP of function f, and m' the number of cubes in the ISOP of the negated function $!f$.

G. Association of Inequalities to Variables

Before computing the variable weights, the tuple *<variables,inequalities>* associating the variables with some of the inequalities is created in step 7 of the method, presented in Fig. 2. By making so, each variable points to inequalities in which the variable is present on the *greater side*. This relationship is exploited in the weight assignment step, discussed in the next section. However, the function defined by (3), used as example in the description of previous steps, is not actually the most appropriate one to illustrate how this procedure works because the simplified set of inequalities has only one inequality. For a better visualization of this step, Fig. 5 illustrates such a kind of relationship for the following function:

$$f_4 = (x_1 \cdot x_2 \cdot x_3) \vee (x_1 \cdot x_2 \cdot x_4) \vee (x_1 \cdot x_3 \cdot x_4) \vee (x_1 \cdot x_2 \cdot x_5) \vee (x_1 \cdot x_3 \cdot x_5) \vee (x_1 \cdot x_4 \cdot x_5) \vee (x_1 \cdot x_2 \cdot x_6) \vee (x_1 \cdot x_3 \cdot x_6) \vee (x_2 \cdot x_3 \cdot x_4) \vee (x_2 \cdot x_3 \cdot x_5). \quad (4)$$

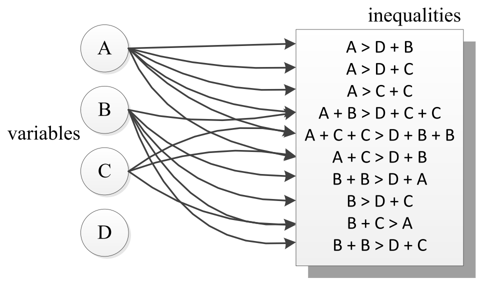


Fig. 5. Example of relationship associating variables and inequalities of function from (4).

H. Variable Weights Assignment

The variable weight assignment step receives the updated set of variables, after the domain transformation, ordered by the VWO parameters, as well as the inequalities and relationships defined in the previous section. The first task is to assign minimum values for each variable. The variable with the lowest VWO parameter value is assigned by 1, the second lowest one by 2, and so on. In the example in Table III, the initial weights assignment trial is $C = 1, B = 2$, and $A = 3$. This step is described in line 1 in Algorithm 8.

The algorithm iterates over all variables, in ascending order (according to the VWO ordering). Each variable points to a set of inequalities, as explained in step 7. The consistency of each inequality is verified, being performed by checking whether the sum of the current values of the *greater side* variables is greater than the sum of the current values of the *lesser side* variables.

The difference between the two inequality sides is called *delta.* If delta is not positive, the inequality is not consistent, i.e., the current assigned weights do not satisfy this inequality. In this case, the value of the variable under verification is incremented, trying to make it valid. Delta is computed in line 6 in Algorithm 8, whereas the consistency is checked in line 7.

When the value of this variable is incremented, the value of the variables with greater VWO parameter is also be incremented in order to maintain the ordering. For instance, considering the following case $(A + C) > (B + B + B)$, increasing the value of C would never turn the inequality consistent because it also increases the values of A and B, i.e., $(A+1+C+1) > (B+1+B+1+B+1)$. In this sense, the *lesser side* cannot increase more than the *greater side*. The procedure *increment_weights* is responsible for incrementing the weight of variable v_i and the weight of the variables greater than v_i.

The decision whether the weight of a variable should be incremented is computed in three steps.

1) Increment the value of the variable, as well as the value of the greater variables, by 1.

Algorithm 8 Weight Assignment

Input: List of *variables list_variables,* relationship of *variables*
Output: Array of weights W with assigned values

```
1:  for all variable $v_i \in list\_variables$ do $W[v_]] = i$
2:  end for
3:  for each $v_i \in list\_variables$ do
4:    $set\_ineq$ = get_ineq_by_variable($v_i$)
5:    for each inequation $ineq \in set\_ineq$
6:      $delta$ = weight_sum($greaterSide$) - weight_sum($lesserSide$)
7:      if ($delta <= 0$) // is the inequality inconsistent?
8:        increment_weights($v_i$,$list\_variables$,1)
9:        $delta'$=weight_sum($greaterSide$)- weight_sum($lesserSide$)
10:       if ($delta' > delta$) //will the variable be incremented?
11:         $increment$ = -$delta$ / ($delta$ - $delta'$)
12:         increment_weights($v_i$,$list\_variables$, $increment$)
13:       else
14:         increment_is_undone
15:   end for
16: end for
17: return $W$
```

TABLE VIII
ORIGINAL SYSTEM, FROM TABLE VI, WITH COMPUTED VARIABLE WEIGHTS

#	Inequality		
1	(3+2)	>	(3+1)
2	(3+2)	>	(3+1)
3	(3+2)	>	(2+1+1)
4	(3+1+1)	>	(3+1)
5	(3+1+1)	>	(3+1)
6	(3+1+1)	>	(2+1+1)

2) Compute the new *delta*, denoted $delta'$·
3) If $delta' = delta$, then the increment is undone and the inequality is kept as inconsistent, and the algorithm proceeds to the next inequalities and variables.

When $delta' > delta$, the *increment_weights* procedure is applied.

The *increment* value that makes the inequality consistent is computed as follows:

$$\text{increment} = \left\lceil \frac{-\text{delta}}{(\text{delta} - \text{delta}')} \right\rceil. \quad (5)$$

Finally, the original system is checked for consistency by replacing the variables by the assigned weights. If all inequalities are consistent, then the values correspond to the right variable weights. Otherwise, if at least one of the inequalities is not consistent then the method identifies the given function as not TLF. This ensures that false positive solutions are not found.

In the previous example, illustrated in Table VII, only the inequality $A > (C+C)$ remains to the assignment step. In this case, the assigned weights are $C = 1, B = 2$, and $A = 3$. The inequality is consistent with these values and so no increment is required. These updated variable values are assigned to the corresponding variable weights, resulting in $w_1 = 3, w_2 = 2$, and $w_3 = w_4 = 1$. Table VIII shows the original system with the assigned values.

Since all inequalities are consistent, the computed values are accepted as a valid solution of the system. Algorithm 8 shows the pseudocode of the assignment task that corresponds to step 8 shown in Fig. 2. The time complexity of this step is $O(n \cdot m \cdot m')$, being n the number of variables, m the number of cubes in the ISOP of function f, and m' the number of cubes in the ISOP of the negated function $!f$.

I. Variable Weights Adjustment

When the weight of a variable is incremented, a true inequality can become false. One way to identify and to prevent the occurrence of such a problem is discussed below.

For a given function, consider that the growing ordering of the updated variable is D, C, B, and A, where A has the highest weight value and D has the lowest weight value. At a certain time, the algorithm has checked the inequalities associated with D, C, and B variables. The inequalities associated with the variable A are checked again, and any inconsistency determines that the value of this variable needs to be incremented. However, there is a certain inequality that has already been checked, e.g., $(B + C + D) > A$. By increasing the value of variable A, the inequality that had been verified consistent could now become inconsistent.

This kind of problem may occur when a variable on the *lesser side* of the inequality has greater value than any variable value on the *greater side*. In our investigation, it has been observed in less than 2% of the six-input functions and never in functions with fewer variables. As a consequence, in these cases, the minimum weight values are not guaranteed.

In order to solve this problem, we present a relationship represented by the tuple <*variable*,*inequalities*>, where the variables point to inequalities. This relationship is similar to that one presented in Section III-G and is called *reverse relation*. If a variable x is on the *lesser side* and its VWO parameter value is greater than any VWO parameter of the *greater side* variables then this information is stored on the relationship tuple. In the assignment, presented in Section III-H, when the method determines that a variable must be incremented, this reverse relation may also be checked.

When incrementing a variable, the method:
1) checks if there is an inequality associated to this variable in the reverse relation;
2) checks if this inequality becomes inconsistent after incrementing the variable x;
3) (if so) increments the greatest value variable of the *greater side*, and makes the inequality consistent again. This is done recursively for the variables with VWO parameter greater than the one of variable x.

It is important to notice that, whenever the value of some variable is incremented, the value of the higher variables must also be incremented, i.e., the ordering must be maintained. A demonstration of this improvement is shown in Section IV-B.

J. Function Threshold Value Computation

After checking whether the input weights have been assigned correctly, the method calculates the function threshold value in step 10. In a TLF represented through an ISOP form, the sum of weights of the variables present in each product is equal to or greater than the threshold value. Therefore, the threshold value is equal to the least sum of weights of the *greater side* set. In the example defined by (3) and in Table I, the threshold value is 5, obtained from the *greater side* element of any inequality in Table VIII (in this case, each *greater side*s

TABLE IX
VWO PARAMETER VALUES FOR FUNCTION OF (5)

VWO parameter	Variables
2	x_5, x_6
14	x_4, x_3
30	x_2
34	x_1

TABLE X
INEQUALITIES GENERATION FOR FUNCTION DESCRIBED BY (5)

Greater side		*Lesser side*
$w_1 + w_2$		$w_3 + w_4 + w_5 + w_6$
$w_1 + w_3$		$w_2 + w_5 + w_6$
$w_1 + w_4$	> T >	$w_1 + w_6$
$w_2 + w_3$		$w_1 + w_5$
$w_2 + w_4$		---
$w_1 + w_5 + w_6$		---

have equal weight sum). The final solution for this particular function is [3, 2, 1, 1; 5].

IV. CASE STUDIES

In the previous section, the method proposed for TLF identification was described. The example of (3), used to demonstrate the algorithms, is quite straightforward, having been adopted just to simplify the explanation and to facilitate the understanding of the procedure steps. However, the importance and impact of each step can be underestimated with such a simple example. In this section, two more complex functions are used as case studies in order to illustrate the major gains and benefits of the proposed method.

A. First Case Study

Consider the following Boolean function:

$$f_5 = (x_1 \cdot x_2) \vee (x_1 \cdot x_3) \vee (x_1 \cdot x_4) \vee (x_2 \cdot x_3) \vee (x_2 \cdot x_4) \vee (x_1 \cdot x_5 \cdot x_6). \tag{6}$$

The first step is to compute the VWO parameter for each variable and sort them, as shown in Table IX.

The given function described in (6) presents 64 possible assignment vectors since it has six variables. Among these assignments, 23 are false and 41 are true. If all truth table assignments were taken into account, then the system would have 943 inequalities. However, as it was discussed before, the method considers the ISOP expression as input and generates only the greatest false assignments and the least true assignments. The *greater side* and *lesser side* sets for this case are presented in Table X. Afterwards, the Cartesian product between the *greater side* and the *lesser side* is performed and 24 inequalities are obtained, as shown in Table XI. In the next, the algorithm creates the updated variables based on repeated VWO parameters and replaces the variable weights, as shown in Table XII. The simplification is then performed by removing variables that appear on both sides of the inequalities. The simplified set of inequalities is presented in Table XIII.

For the variable weight assignment, the method selects only inequalities that have more than one weight on the *lesser side* and discards repeated inequalities. Table XIV presents the selected inequalities for the given example. We have only 8 inequalities against 943 inequalities whether a conventional method had been applied and against 24 inequalities present in the system originally generated.

TABLE XI
ORIGINAL INEQUALITIES SYSTEM FOR FUNCTION DESCRIBED BY (5)

#	Inequality	#	Inequality
1	$w_1+w_2>w_3+w_4+w_5+w_6$	13	$w_2+w_3>w_3+w_4+w_5+w_6$
2	$w_1+w_2>w_2+w_5+w_6$	14	$w_2+w_3>w_2+w_5+w_6$
3	$w_1+w_2>w_1+w_6$	15	$w_2+w_3>w_1+w_6$
4	$w_1+w_2>w_1+w_5$	16	$w_2+w_3>w_1+w_5$
5	$w_1+w_3>w_3+w_4+w_5+w_6$	17	$w_2+w_4>w_3+w_4+w_5+w_6$
6	$w_1+w_3>w_2+w_5+w_6$	18	$w_2+w_4>w_2+w_5+w_6$
7	$w_1+w_3>w_1+w_6$	19	$w_2+w_4>w_1+w_6$
8	$w_1+w_3>w_1+w_5$	20	$w_2+w_4>w_1+w_5$
9	$w_1+w_4>w_3+w_4+w_5+w_6$	21	$w_1+w_5+w_6>w_3+w_4+w_5+w_6$
10	$w_1+w_4>w_2+w_5+w_6$	22	$w_1+w_5+w_6>w_2+w_5+w_6$
11	$w_1+w_4>w_1+w_6$	23	$w_1+w_5+w_6>w_1+w_6$
12	$w_1+w_4>w_1+w_5$	24	$w_1+w_5+w_6>w_1+w_5$

TABLE XII
UPDATED VARIABLE CREATED FOR EACH VWO PARAMETER VALUE FOR FUNCTION DESCRIBED BY (5)

Variable weights	VWO parameter	Updated variable
w_1	34	A
w_2	30	B
w_3, w_4	14	C
w_5, w_6	2	D

TABLE XIII
SIMPLIFIED INEQUALITIES SYSTEM FROM TABLE XI USING UPDATED VARIABLES FROM TABLE XII

#	Inequality			#	Inequality		
1	A+B	>	C+C+D+D	13	B	>	C+D+D
2	A	>	D+D	14	C	>	D+D
3	B	>	D	15	B+C	>	A+D
4	B	>	D	16	B+C	>	A+D
5	A	>	C+D+D	17	B	>	C+D+D
6	A+C	>	B+D+D	18	C	>	D+D
7	C	>	D	19	B+C	>	A+D
8	C	>	D	20	B+C	>	A+D
9	A	>	C+D+D	21	A	>	C+C
10	A+C	>	B+D+D	22	A	>	B
11	C	>	D	23	D	>	0
12	C	>	D	24	D	>	0

Fig. 6 illustrates the variable weight assignment. At first, the variables are assigned with minimum values, respecting the ordering defined by the respective VWO parameters. Thus, these values are $A = 4$, $B = 3$, $C = 2$, and $D = 1$, as indicated in Fig. 6 (1). Since there are not any inequalities pointed by variable D, the method accepts the value 1 for this variable and starts the verification of the inequalities pointed by C. This way, the inequality #14 in Table XIV is not consistent. This inequality becomes consistent by increasing the value of variable C, as indicated in Fig. 6 (2). The updated value of C is 3, and all of the inequalities containing variable C are now consistent.

In the next, the method checks the inequalities pointed by B. The verification concludes that the inequality #13 in Table XIV

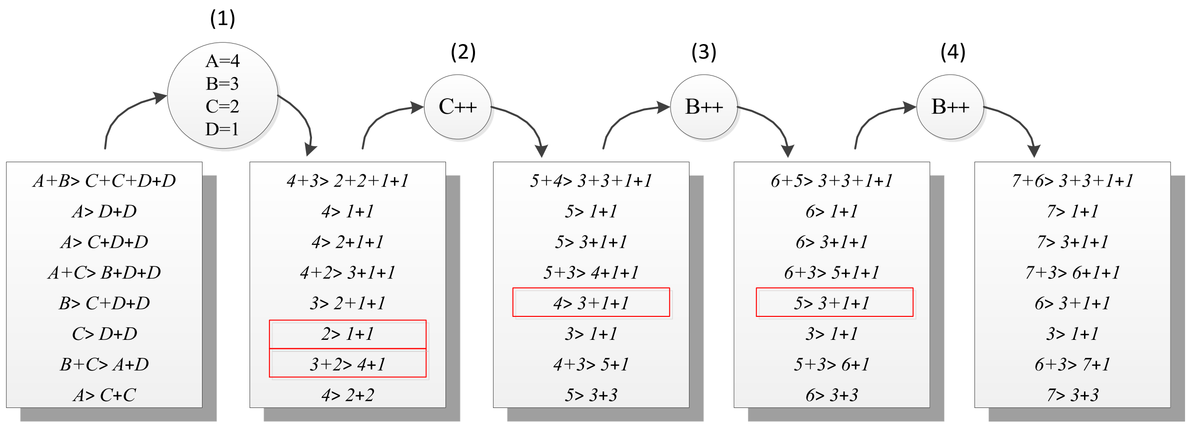


Fig. 6. Variable weight assignment for the first case study described by (5).

TABLE XIV
SELECTED INEQUALITIES FROM TABLE XIII FOR THE VARIABLE WEIGHT ASSIGNMENT STEP

#	Inequality		
1	A+B	>	C+C+D+D
2	A	>	D+D
5	A	>	C+D+D
6	A+C	>	B+D+D
13	B	>	C+D+D
14	C	>	D+D
15	B+C	>	A+D
21	A	>	C+C

is inconsistent. However, by incrementing just once the value of variable B does not become the inequality valid, as indicated in Fig. 6 (3). Instead, it decreases the difference between the sums of the two sides.

The increment is performed again, and the current values are $B = 6$ and $A = 7$, as indicated in Fig. 6 (4), now making all inequalities pointed by B consistent. Finally, the method checks the inequalities pointed by A and verifies that these inequalities are also valid with the weights $A = 7$, $B = 6$, $C = 3$, and $D = 1$.

After computing all variable weights, the algorithm defines the function threshold value. This value is obtained from the least sum of weights value in the *greater side* set. In this case, the threshold value is equal to 9 in any of the weight sums in Table XI, $(w_2 + w_4)$, $(w_2 + w_3)$, or $(w_1 + w_5 + w_6)$.

The final check is then performed over the original inequalities system presented in Table XI, i.e., before the simplification step, with the assigned variable weights. All inequalities are verified consistent, so the right solution found is $w_1 = 7$, $w_2 = 6$, $w_3 = 3$, $w_4 = 3$, $w_5 = 1$, $w_6 = 1$, and $T = 9$.

Due to the bottom-up characteristic of our approach, the variable weights and function threshold value found are the minimum possible ones. This is a quite relevant feature because it impacts directly on the circuit area of the corresponding TLG. In the Palaniswamy *et al.*'s [13] method, for instance, the solution found for the same TLF is [9, 8, 4, 4, 1, 1; 11].

B. Second Case Study

The second case study represents a TLF that is not found by other related heuristic approaches. It demonstrates the improvement discussed in Section III-I, exploiting the reverse relation concept.

TABLE XV
UPDATED VARIABLE CREATED FOR EACH VWO PARAMETER VALUE OF FUNCTION DESCRIBED BY (6)

Variable weights	VWO parameter	Updated variable
w_1, w_2	34	A
w_3, w_4, w_5	30	B
w_6	14	C

TABLE XVI
SIMPLIFIED INEQUALITIES FOR THE WEIGHT ASSIGNMENT STEP OF FUNCTION DESCRIBED BY (6)

#	Inequality		
1	B+B	>	A
2	A	>	C+B
3	A+A	>	B+B+B
4	A+A	>	C+B+B

Consider the function represented by the following ISOP:

$$f_6 = (x_1 \cdot x_2 \cdot x_3) \vee (x_1 \cdot x_2 \cdot x_4) \vee (x_1 \cdot x_3 \cdot x_4) \vee (x_1 \cdot x_2 \cdot x_5) \vee (x_1 \cdot x_3 \cdot x_5) \vee (x_1 \cdot x_4 \cdot x_5) \vee (x_1 \cdot x_2 \cdot x_6) \vee (x_2 \cdot x_3 \cdot x_4) \vee (x_2 \cdot x_3 \cdot x_5) \vee (x_2 \cdot x_4 \cdot x_5). \quad (7)$$

Table XV shows the relationship between the variable weights, the corresponding VWO parameters, and the created updated variables. For the sake of simplicity, the process of generation and simplification of variables is not presented. Table XVI shows the already simplified system using the updated variables.

As discussed in Section III-G, for this case, the relationship associating the inequalities of the system with variables that appear in the *greater side* would be the variable B pointing to inequality #1 and the variable A pointing to inequalities #2–#4 in Table XVI.

Inequality #1, shown in Table XVI, presents the characteristics described in Section III-I, where the variable with the greatest value appears on the *lesser side* of the inequality. This information is stored in the reverse relation, with variable A pointing to this inequality. In other words, when it is necessary to increase the value of variable A, it is required to check whether this inequality has not become inconsistent. If

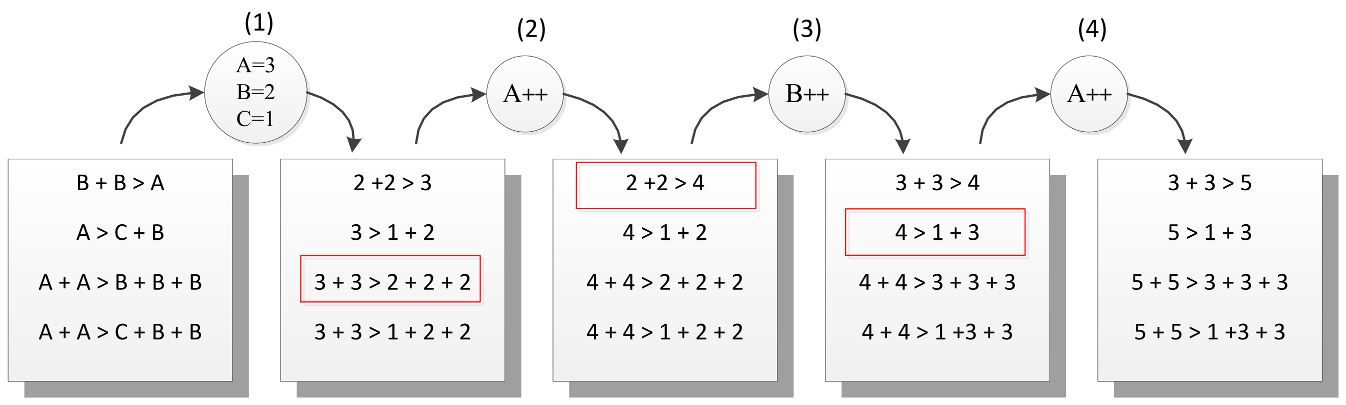


Fig. 7. Variable weight assignment for the second case study described by (6).

TABLE XVII
NUMBER OF TLFS IDENTIFIED BY EACH METHOD

Number of variables	Muroga [23]	Gowda [10]		Palaniswamy [13]		Proposed method	
	TLF count	TLF count	%	TLF count	%	TLF count	%
1	2	2	100	2	100	2	100
2	8	8	100	8	100	8	100
3	72	72	100	72	100	72	100
4	1,536	1,248	81.3	1,536	100	1,536	100
5	86,080	36,800	42.8	75,200	87.4	**86,080**	**100**
6	14,487,040	1,447,040	10.0	9,424,000	65.1	**14,487,040**	**100**

the inequality is now inconsistent, the variable with the greatest value of *greater side*, in this case, the variable B, must also be incremented.

The weight assignment process occurs as shown in Fig. 7. First of all, the variables are assigned to minimum weights, such as $C = 1, B = 2$, and $A = 3$, as indicated in Fig. 7 (1). Since there is not any inequality to be checked associated with variable C, the value 1 is accepted. To check the current value of variable B, the inequality $(B+B) > A$ is verified and, at this moment, it is consistent. Thus, the value 2 is accepted for B.

Finally, the inequalities pointed by variable A are verified. The inequality $(A+A) > (B+B+B)$ is inconsistent. Increasing the value of variable A would make valid the inequality, as indicated in Fig. 7 (2). However, since there is the inequality $(B + B) > A$ in the reverse relation, it is necessary to verify if this inequality remains valid. With $B = 2$ and $A = 4$, the inequality is inconsistent. This means that when the process increments the value of variable A, the value of B must also be incremented becoming equal to 3, as indicated in Fig. 7 (3).

The current values are $A = 4, B = 3$, and $C = 1$. Looking at the inequalities pointed by variable A, the method checks that the inequality $A > (B + C)$ is inconsistent. Hence, the value of the variable A is incremented again and the inequality becomes valid. At this time, the increment does not make inconsistent the inequality $(B + B) > A$, as indicated in Fig. 7 (4). The new value of variable A is accepted and the solution found is $A = 5, B = 3$, and $C = 1$. When replacing these values in the original variable weights, it is possible to check the consistency of the original system with all inequalities. Therefore, the solution is valid. The final solution is $w_1 = w_2 = 5, w_3 = w_4 = w_5 = 3$, and $w_6 = 1$, and the function threshold value is equal to 11, also represented by [5, 5, 3, 3, 3, 1; 11].

V. EXPERIMENTAL RESULTS

Three sets of experiments were carried out in order to validate our approach in comparison to the related methods available in the literature. In the first experiment, the effectiveness of the proposed method is evaluated in terms of the number of TLF identified. In the second experiment, the functions extracted from k-cut enumeration in the logic synthesis of opencores are analyzed for both effectiveness and runtime of TLF identification. The execution time is verified in the third experiment, calculating the runtime per function identified and estimating the method scalability. The platform used to run the experiments was an Intel Core i5-2400 processor with 8 GB of main memory. The ILP solver adopted was the *lpsolve* 5.5 [29].

A. *Identification Effectiveness*

The main goal of TLF identification methods is to maximize the number of TLFs identified, representing their effectiveness. The enumeration of all TLFs with up to eight variables has been already calculated by Muroga *et al.* [23], being used as a reference in this paper.

Gowda *et al.* [10] presented the first approach to identify TLF without using linear programming, afterward improved in [11]. The method presented by Palaniswamy *et al.* [13], provides better results than the Gowda *et al.*'s [10] one.

Table XVII shows the experimental results. For functions with up to three variables, both Gowda *et al.*'s [10] and Palaniswamy *et al.*'s [13] methods identify 100% of existing TLFs. Moreover, the Gowda *et al.*'s [10] method identifies 81.3%, 42.8%, and 10% of TLFs with four, five, and six variables, respectively, whereas the Palaniswamy *et al.*'s [13] identified 100%, 87.4%, and 65.1%, respectively, for the same sets. Our method identified all existing TLFs with up to five

TABLE XVIII
NUMBER OF TLFS IDENTIFIED BY EACH METHOD CONSIDERING NP REPRESENTATIVE CLASS OF FUNCTIONS

Number of variables	Muroga [23] TLF count	Gowda [10] TLF count	Gowda [10] %	Palaniswamy [13] TLF count	Palaniswamy [13] %	Proposed method TLF count	Proposed method %
1	1	1	100	1	100	1	100
2	2	2	100	2	100	2	100
3	5	5	100	5	100	5	100
4	17	15	88.2	17	100	17	100
5	92	52	56.5	84	91.3	92	100
6	994	181	18.2	728	73.2	994	100
7	28,262	573	2.0	9,221	32.6	**22,477**	**79.5**

TABLE XIX
ANALYSIS OF k-CUTS (n-VARIABLE FUNCTIONS) OBTAINED FROM THE SYNTHESIS OF ALTERA OPENCORES [25]

cuts	#Functions (k-cuts)	Unate (unate/total)	TLF - ILP (TLF/unate)	TLF - proposed (proposed/ILP)	ILP runtime (sec)	proposed runtime (sec) (Speed up)
k-6	32,137	863 (3%)	306 (35%)	306 (100%)	0.07	0.01 (7.0)
k-7	119,654	3673 (3%)	822 (22%)	817 (99%)	0.35	0.04 (9.8)
k-8	283,005	11192 (4%)	1786 (16%)	1758 (98%)	1.17	0.12 (9.8)
k-9	529,570	22980 (4%)	3238 (14%)	3183 (98%)	2.65	0.27 (9.8)
k-10	835,692	38469 (5%)	4853 (13%)	4766 (98%)	4.87	0.51 (9.5)
k-11	1,172,438	56321 (5%)	6752 (12%)	6645 (98%)	7.68	0.87 (8.8)
k-12	1,531,068	75243 (5%)	9516 (13%)	9390 (99%)	11.27	1.32 (8.5)
k-13	1,813,459	96294 (5%)	13041 (14%)	12907 (99%)	15.50	2.04 (7.6)
k-14	2,171,666	112070 (5%)	15166 (14%)	15020 (99%)	19.64	2.86 (6.9)
k-15	2,349,808	123359 (5%)	17698 (14%)	17544 (99%)	23.49	3.82 (6.1)

variables and, to the best of our knowledge, this is the first non-ILP-based method able to identify all TLFs with six variables.

Since the number of TLFs with seven inputs is too large, corresponding to 8 274 794 440 functions, we have adopted the NP representative class of functions [18]. The results are shown in Table XVIII. Notice that, for the universe of TLFs with seven variables, our approach identified almost 80% of them, whereas the other methods identify less than 35%.

B. Altera Opencores

In the second experiment, we aim to demonstrate the runtime efficiency when testing a very large number of small subcircuits. That is a typical task performed by technology mapping tools, where the goal is to match, as quickly as possible, the functions obtained during the decomposition process with an implementation in a single threshold gate. In this sense, we have enumerated all the priority cuts of the opencores [25].

The cut enumeration was performed by the technology mapper available in the ABC logic synthesis tool [26]. In general, during the matching process, the entire set of enumerated functions has no more than 16 inputs. Notice that ABC tool limits the k-cuts to 15 inputs due to internal restrictions of truth table representation. In this experiment, our method was able to identify more than 2 million functions with up to 15 inputs in approximately 11 s.

Table XIX summarizes the number of TLFs found by both proposed approach and ILP-based method. In the third column, the amount and respective percentage of unate functions among the k-cuts obtained from the synthesis of the opencores are presented. Remember that only unate functions are candidates to be TLF. The amount of TLFs in the unate functions set identified by the ILP-based method represents the total number of TLFs, as shown in the fourth column in Table XIX. The number of TLFs identified by our method is shown in the fifth column. The sixth and seventh columns present the total execution time for the identification of all k-cuts with up to n inputs.

These experimental results demonstrate a good tradeoff between the number of TLFs identified and the corresponding runtime when applying the proposed method. The execution time was improved more than $8\times$ in average when compared to the ILP-based method. Moreover, our approach does not show a loss in the quality of the results once the number of identified TLFs is practically the totality of them. The method described in this paper is available in the current version of the ABC logic synthesis tool. It has been applied in a complete threshold logic synthesis flow described in [15].

C. Runtime Efficiency

In the third experiment, the scalability of our heuristic approach was evaluated. The method is able to identify TLF either from a truth table or from an ISOP expression. In general, the truth table representation is not suitable to represent Boolean functions with more than 16 variables. Therefore, in order to demonstrate the method scalability, we have generated ISOP forms of TLFs with a considerable number of inputs.

The time complexity of the proposed method depends mainly on two values: 1) the number of cubes on the ISOP representation of the input function and 2) the maximal input weight value.

1) Functions With Huge Number of ISOP Cubes: We have evaluated the method considering also a given system with a huge number of inequalities. The number of inequalities (i.e., the system size) is equal to $|f| \cdot |!f|$, where $|f|$ and $|!f|$ represent the number of cubes in the ISOP of function f and the corresponding negated function $!f$, respectively. The majority

Boolean function MAJ(n), or voter, has an ISOP representation with a very large number of cubes: MAJ(n) function has $(nn/2)$ cubes.

2) TLF With Very Large Input Weights: In order to evaluate our method in identifying threshold functions with very large input weights, we have recursively defined a class of function, called herein as the onion(n) functions, as follows:

$$\text{onion}(n) = \begin{cases} x_i, & n = 1 \quad (8a) \\ x_i \cdot (\text{onion}(n-1)), & n \text{ is odd} \quad (8b) \\ x_i \vee (\text{onion}(n-1)), & n \text{ is even.} \quad (8c) \end{cases}$$

For instance, take a look on the following functions:

$$\text{onion}(5) = (x_5 \cdot (x_4 \cdot (x_3 \cdot (x_2 \vee x_1)))) \quad (9a)$$

$$\text{onion}(8) = (x_8 \vee (x_7 \cdot (x_6 \vee (x_5 \cdot (x_4 \vee (x_3 \cdot (x_2 \vee x_1))))))). \quad (9b)$$

Onion(n) functions are TLF by definition [19]. Besides, during the experiments, we have figured out that this class of function presents an interesting property: the weight w_i of a variable i is equivalent to the ith number of the Fibonacci sequence, presenting a well-known behavior. Therefore, we have used the onion(n) functions to generate TLF with a larger number of inputs and a very large maximal weight.

3) Balancing the Number of ISOP Cubes and the Input Weight: We have defined a class of functions, called MAJion(n), as follows:

$$\text{MAJion}(n) \begin{cases} x_i, & n = 1 \quad (10a) \\ x_n \cdot (\text{MAJion}(n-1)), & n\%3 = 0 \quad (10b) \\ x_n \vee (\text{MAJion}(n-1)), & n\%3 = 1 \quad (10c) \\ \text{MAJ}(x_n, x_{n-1}, \text{MAJion}(n-2)), & n\%3 = 2 \quad (10d) \end{cases}$$

where % represents the modulo operation.

For instance, take a look on the following functions:

$$\text{MAJion}(5) = \text{MAJ}(x_5, x_4, (x_3 \cdot (x_2 \vee x_1))) \quad (11a)$$

$$\text{MAJion}(9) = \text{MAJ}(x_9, x_8, (x_7 \cdot (x_6 \vee (\text{MAJ}(x_5, x_4, (x_3 \; (x_2 \vee x_1))))))). \quad (11b)$$

MAJion(n) functions are similar to onion(n) functions. However, MAJion(n) has also MAJ(n) functions in order to increase the ISOP cubes count. MAJion(n) functions have a smaller number of distinct VWO but more cubes when compared to the onion(n) functions, and a larger number of distinct VWO but fewer cubes when compared to the MAJ(n) ones. In this sense, we believe that the class of MAJion(n) functions fits well to a balanced tradeoff between the cubes count and the largest input weight.

The experimental results regarding the threshold identification for onion(n) and MAJion(n) functions are shown in Table XX. These results demonstrate that for onion(n) functions both proposed method and ILP-based approach are very efficient due to the small number of inequalities, even taking into account very high maximal weights. For MAJion(n) from 50 to 65 inputs, the proposed method runs in less than 1 s, whereas the ILP-based is several orders of magnitude slower. For MAJion(n) with more than 65 inputs, the ILP-based method was not able to solve with a timeout about 10 000 s.

The limits of our implementation when running MAJ(n), onion(n), and MAJion(n) functions are as follows.

1) The number of cubes for MAJ(n) functions becomes too large. For instance, the input file containing the ISOP of MAJ(27) has 1.2 GB of data. In this case, our method runs in 10.1 s. We were not able to run the method for the MAJ(31) function since the input file has 20 GB of data.
2) The number of cubes for onion(n) functions, on the other hand, grows linearly but the worst input weight grows exponentially to the number of inputs. In this sense, we were able to obtain results for the onion(n) functions with up to 92 inputs. In particular, for onion(92), our method runs in 1.2 ms, being the worst input weight equal to 7.5×10^{18}. The worst input weight for onion(n) functions with more than 92 inputs cannot be represented through an unsigned integer of 64 bits.
3) The cubes count on MAJion(n) grows slower than on MAJ(n) but faster than on onion(n) functions. We were able to obtain results for MAJion(n) functions with up to 70 inputs. In this case, the number of cubes is more than 500 000 and the maximal input weight is more than 6×10^{12}.

TABLE XX
TLF IDENTIFICATION FOR LARGE ISOPS

Function	#inputs (n)	#cubes (ISOP)	max weight	runtime (proposed)	runtime (ILP)
MAJ(n)	15	6,435	1	0.003 s	0.22 s
	17	24,310	1	0.009 s	2.74 s
	19	92,378	1	0.04 s	37.6 s
	21	352,716	1	0.15 s	588 s
	23	1,352,078	1	0.61 s	887 s
onion(n)	65	33	~832 10^3	0.4 ms	1.4 ms
	70	36	~102 10^6	0.5 ms	1.6 ms
	75	38	~12.6 10^9	0.6 ms	1.8 ms
	80	41	~2.34 10^{15}	0.7 ms	2.0 ms
	85	43	~421 10^{15}	0.9 ms	2.3 ms
MAJion(n)	50	16,382	~1.12 10^9	0.02 s	5.6 s
	55	32,776	~17.8 10^9	0.05 s	22.5 s
	60	65,535	~101 10^9	0.13 s	340 s
	65	262,141	~1.14 10^{12}	0.61 s	> 10000 s
	70	524,286	~6.42 10^{12}	1.35 s	> 10000 s

VI. CONCLUSION

A straightforward and very effective non-ILP-based method for identifying TLFs was presented. The variable weights are assigned using a bottom-up strategy based on the VWO parameter ordering. As demonstrated by experimental results, the method identifies more TLFs than other existing related heuristic approaches, and the obtained solutions are minimum in terms of variable weight for 98% of cases. Although representing Boolean functions is intrinsically an exponential problem, we showed the runtime is scalable, enabling the application of the method when the number of variables increases.

REFERENCES

[1] (2011). *Semiconductor Industries Association Roadmap*. [Online]. Available: http://public.itrs.net

[2] R. Zhang, P. Gupta, L. Zhong, and N. K. Jha, "Threshold network synthesis and optimization and its application to nanotechnologies," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 24, no. 1, pp. 107–118, Jan. 2005.

[3] N. S. Nukala, N. Kulkarni, and S. Vrudhula, "Spintronic threshold logic array (STLA)—A compact, low leakage, non-volatile gate array architecture," in *Proc. IEEE/ACM Int. Symp. Nanoscale Archit.*, Amsterdam, The Netherlands, 2012, pp. 188–195.

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

[5] L. Gao, F. Alibart, and D. B. Strukov, "Programmable CMOS/memristor threshold logic," *IEEE Trans. Nanotechnol.*, vol. 12, no. 2, pp. 115–119, Mar. 2013.

[6] V. Beiu, J. M. Quintana, and M. J. Avedillo, "VLSI implementations of threshold logic-a comprehensive survey," *IEEE Trans. Neural Netw.*, vol. 14, no. 5, pp. 1217–1245, Sep. 2003.

[7] A. K. Maan, D. A. Jayadevi, and A. P. James, "A survey of memristive threshold logic circuits," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 28, no. 8, pp. 1734–1746, Aug. 2016.

[8] C. B. Dara, T. Haniotakis, and S. Tragoudas, "Delay analysis for current mode threshold logic gate designs," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 25, no. 3, pp. 1063–1071, Mar. 2016.

[9] J. L. Subirats, J. M. Jerez, and L. Franco, "A new decomposition algorithm for threshold synthesis and generalization of Boolean functions," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 55, no. 10, pp. 3188–3196, Nov. 2008.

[10] T. Gowda, S. Vrudhula, and G. Konjevod, "A non-ILP based threshold logic synthesis methodology," in *Proc. Int. Workshop Logic Syn.*, 2007, pp. 222–229.

[11] T. Gowda, S. Vrudhula, N. Kulkarni, and K. Berezowski, "Identification of threshold functions and synthesis of threshold networks," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 30, no. 5, pp. 665–677, May 2011.

[12] A. K. Palaniswamy, M. K. Goparaju, and S. Tragoudas, "Scalable identification of threshold logic functions," in *Proc. Great Lakes Symp. VLSI*, Providence, RI, USA, 2010, pp. 269–274.

[13] A. K. Palaniswamy, M. K. Goparaju, and S. Tragoudas, "An efficient heuristic to identify threshold logic functions," *ACM J. Emerg. Technol. Comput. Syst.*, vol. 8, no. 3, pp. 1–17, 2012.

[14] A. Neutzling, M. G. A. Martins, R. P. Ribas, and A. I. Reis, "Synthesis of threshold logic gates to nanoelectronics," in *Proc. Symp. Integr. Circuits Syst. Design*, Curitiba, Brazil, 2013, pp. 1–6.

[15] A. Neutzling, J. M. A. Matos, A. I. Reis, R. P. Ribas, and A. Mishchenko, "Threshold logic synthesis based on cut pruning" in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design*, Austin, TX, USA, 2015, pp. 494–499.

[16] N. Kulkarni, J. Yang, J.-S. Seo, and S. Vrudhula, "Reducing power, leakage, and area of standard-cell ASICs using threshold logic flip-flops," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 24, no. 9, pp. 2872–2886, Sep. 2016.

[17] R. K. Brayton, A. L. Sangiovanni-Vincentelli, C. T. McMullen, and G. D. Hatchel, *Logic Minimization Algorithms for VLSI Synthesis*. Norwell, MA, USA: Kluwer Acad., 1984.

[18] U. Hinsberger and R. Kolla, "Boolean matching for large libraries," in *Proc. Design Autom. Conf.*, San Francisco, CA, USA, 1998, pp. 206–211.

[19] S. Muroga, *Threshold Logic and Its Applications*. New York, NY, USA: Wiley, 1971.

[20] S. Minato, "Fast generation of prime-irredundant covers from binary decision diagrams," *IEICE Trans. Fundam. Elect. Commun. Comput. Sci.*, vol. E76-A, no. 6, pp. 967–973, 1993.

[21] G. Birkhoff *et al.*, *Lattice Theory*, vol. 25. New York, NY, USA: Amer. Math. Soc., 1948.

[22] R. O. Winde, "Chow parameters in threshold logic," *J. ACM*, vol. 18, no. 2, pp. 265–289, 1971.

[23] S. Muroga, T. Tsuboi, and C. R. Baugh, "Enumeration of threshold functions of eight variables," *IEEE Trans. Comput.*, vol. C-19, no. 9, pp. 818–825, Sep. 1970.

[24] E. M. Stentovich *et al.*, "SIS: A system for sequential circuit synthesis," EECS Dept., Univ. California, at Berkeley, Berkeley, CA, USA, Tech. Rep. UCB/ERL M92/41, 1992.

[25] J. Pistorius, M. Hutton, A. Mishchenko, and R. Brayton, "Benchmarking method and designs targeting logic synthesis for FPGAs," in *Proc. Int. Workshop Logic Syn.*, 2007, pp. 230–237.

[26] Berkeley Logic Synthesis and Verification Group. *ABC: A System for Sequential Synthesis and Verification Release 20160425*. Accessed on Jul. 6, 2017. [Online]. Available: https://people.eecs.berkeley.edu/∼alanmi/abc/

[27] Y.-C. Chen, R. Wang, and Y.-P. Chang, "Fast synthesis of threshold logic networks with optimization," in *Proc. Asia South Pac. Design Autom. Conf.*, Macau, China, 2016, pp. 486–491.

[28] M. Berkelaar, K. Eikland, and P. Notebaert. *Lp_Solve 5.5, Open Source (Mixed-Integer) Linear Programming System. Release 5.5*. Accessed on Jul. 6, 2017. [Online]. Avaliable: http://lpsolve.sourceforge.net/5.5/

[29] M. K. Goparaju and S. Tragoudas, "A fault tolerant design methodology for threshold logic gates and its optimizations," in *Proc. IEEE Int. Symp. Qual. Elect. Design*, San Jose, CA, USA, 2007, pp. 420–425.

[30] A. K. Palaniswamy, S. Tragoudas, and T. Haniotakis, "ATPG for delay defects in current mode threshold logic circuits," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 35, no. 11, pp. 1903–1913, Nov. 2016.

Augusto Neutzling (S'13) received the B.S. degree in computer engineering from the Federal University of Rio Grande, Rio Grande, Brazil, in 2012 and the M.S. degree in computer science from the Federal University of Rio Grande do Sul, Porto Alegre, Brazil, in 2014, where he is currently pursuing the Ph.D. degree.

Mayler G. A. Martins (M'16) received the B.S. degree in computer engineering from the Federal University of Espírito Santo, Vitória, Brazil, in 2009, and the M.S. (*summa cum laude*) and Ph.D. degrees in microelectronics from the Federal University of Rio Grande do Sul, Porto Alegre, Brazil, in 2012 and 2015, respectively.

He is currently a Researcher with Carnegie Mellon University, Pittsburgh, PA, USA.

Vinicius Callegaro (S'14–M'17) received the B.S., M.S. *(summa cum laude)*, and Ph.D. *(summa cum laude)* degrees in computer science from the Federal University of Rio Grande do Sul (UFRGS), Porto Alegre, Brazil, in 2009, 2012, and 2016, respectively.

From 2016 to 2017, he was a Post-Doctoral Researcher with PGMICRO, UFRGS. He is currently a Senior Research and Development Engineer with the ICDS Synthesis Solutions Group, Mentor-A Siemens Business, Fremont, CA, USA.

André I. Reis (M'99–SM'05) received the B.S. degree in electrical engineering from the Federal University of Rio Grande do Sul (UFRGS), Porto Alegre, Brazil, in 1991 and the Ph.D. degree in automatic and microelectronics systems from UMII, Montpellier, France, in 1998.

He has been a Professor with the Institute of Informatics, UFRGS, since 2000.

Renato P. Ribas (M'12) received the B.S. degree in electrical engineering from the Federal University of Rio Grande do Sul (UFRGS), Porto Alegre, Brazil, in 1991 and the Ph.D. degree in microelectronics from the Institut National Polytechnique de Grenoble, Grenoble, France, in 1998.

He has been a Professor with the Institute of Informatics, UFRGS, since 2000.