Integrated ESOP Refactoring for Industrial Designs

Winston Haaswijk*, Luca G. Amarù†, Patrick Vuillod†, Jiong Luo†, Mathias Soeken*, Giovanni De Micheli*
*Integrated Systems Laboratory, EPFL, Switzerland †Synopsys Inc., CA, USA

***Abstract*—We present a multi-level logic refactoring algorithm based on *exclusive sum-of-product* (ESOP) expressions. ESOP expressions are two-level logic representation forms, similar to *sum-of-product* (SOP) expressions. However, ESOPs use EXOR instead of OR operators. It has been shown that this allows ESOPs to be exponentially more compact than SOP expressions for important classes of functions. Our algorithm is based on a combination of ESOP collapsing, minimization, and refactoring. In EXOR-heavy logic, such as arithmetic units, it unlocks optimizations that may be outside the reach of SOP based methods. We show that our method is able to significantly improve upon logic optimization results, as compared to a similar SOP based flow. On a set of EXOR-heavy benchmarks, we reduce logic levels by up to 83.3% in the best case, and by 44.6% on average. Further, we are able to reduce logic network size by 21.4% on average. We have integrated our method into a commercial synthesis flow. On a set of 46 industrial benchmarks, the optimizations introduced by our algorithm improve design results after physical synthesis.**

I. Introduction

The two-level *sum-of-products* (SOP) representation is perhaps the most widely used way to represent Boolean functions. However, it is not necessarily the most efficient one. The main object of interest in this paper is the *exclusive-sum-of-products* (ESOP) representation. ESOPs differ from the SOPs in that they use the EXOR instead of the OR operator. Both SOP and ESOP expressions can be used to represent any Boolean function. However, ESOPs have the theoretical advantage of being more compact for important classes of functions [1]. For example, the parity function of 4 variables can be written as the ESOP expression $a \oplus b \oplus c \oplus d$, whereas the smallest possible SOP expression requires 8 product terms and 32 literals:

$$abc\bar{d} + ab\bar{c}d + a\bar{b}cd + \bar{a}bcd + a\bar{b}\bar{c}\bar{d} + \bar{a}b\bar{c}\bar{d} + \bar{a}\bar{b}cd + \bar{a}\bar{b}\bar{c}d$$

ESOP *minimization* refers to the process of finding a minimal ESOP representation for a given function. There is no known efficient algorithm for finding the exact minimum ESOP expression for arbitrary functions. Currently, finding such minima can only be done for arbitrary functions up to 6 variables, or up 10 variables for specific classes of functions [2], [3], [4]. Therefore, various types of heuristic ESOP minimization have been developed. Most recent attempts at heuristic minimization algorithms are based upon the iterative application of cube transformations [5], [6], [7], [8], [1], [9], [10]. The *ExorLink* operation, introduced by Song and Perkowski in [8], generalized various other cube transformations (e.g. the *xlinking* and *unlinking* operations) and integrated them into a single transformation [5], [6], [7]. Finally, ESOP minimization is harder than SOP minimization [1], due to the fact that ESOP minimization is a binate covering problem, whereas SOP minimization is a unate covering problem [11, pp. 268][12, pp. 278].

There is also a noteable literature on EXOR based logic minimization using *sum-of-pseudoproduct* (SPP) expressions [13], [14]. SPPs are three-level logic expressions in which the two-level concept of cubes is generalized to *pseudocubes*, which are products of EXOR factors. A k-SPP is an SPP that has EXORs with k-bounded fanin [15]. Three-level forms such as SPPs and k-SPPs have the advantage that they can represent functions more compactly than two-level forms such as (E)SOPs. Exact and heuristic algorithms to minimize such expressions have been developed, we refer the interested reader to [13], [15], [14].

Besides theoretical and academic motivations, a key driving factor in the progress of ESOP minimization has been the construction of *programmable logic arrays* (PLAs), which rely on EXOR rather than OR arrays. EPLAs are easier to test and require less hardware [1]. Further, for multi-level logic networks, the use of EXOR gates can lead to the synthesis of circuits with less area and delay [16].

For circuit designs that contain EXOR-heavy logic, conventional methods based on SOP expressions may miss some optimizations, as they cannot find compact EXOR representations. Motivated by the theoretical compactness advantage of ESOPs, our goal in this paper is to develop an ESOP analogue to conventional SOP based collapsing. Such an optimization flow consists of SOP collapsing, minimization, and refactoring. We present an analogous multi-stage ESOP refactoring method, and show that the theoretical advantage of ESOP indeed carries over into significantly improved logic optimization results. Our method is also designed to be integrated into an existing commercial optimization framework that uses window enumeration and truth table computation. Such integration imposes stringent design requirements, leading to the efficient optimization method presented in this paper.

We compare our ESOP optimization method to a similar flow based on SOPs. The comparison reveals significant improvements, reducing logic levels by up to 83.3% and by 44.6% on average. Further, our method reduces average logic network size by 21.4%. Finally, we integrate our method into a commercial synthesis flow. On a set of 46 industrial benchmarks we show average improvement, after placement and routing, of 1.08% in area, 0.93% in leakage power, and 0.15% in worst negative slack.

II. Preliminaries

Given a Boolean function $f : \mathbb{B}^n \to \mathbb{B}$, we write the *cofactor* of f with respect to variable x_i as $f_{\bar{x}_i}$ for $x_i = 0$ and f_{x_i} for $x_i = 1$. We can then define the following *expansions* of the function:

$f = \bar{x}_i f_{\bar{x}_i} \oplus x_i f_{x_i}$	Boole's expansion
$f = f_{\bar{x}_i} \oplus x_i (f_{\bar{x}_i} \oplus f_{x_i})$	positive Davio expansion
$f = f_{x_i} \oplus \bar{x}_i (f_{\bar{x}_i} \oplus f_{x_i})$	negative Davio expansion

A *literal* is typically defined as a Boolean variable x_i in either positive or negative polarity. For example, the variable x_i may be written in negative polarity as $\bar{x}_i$. In the context of ESOP minimization, it is useful to extend this definition to work with *multiple-valued* literals. Let $P_i = \{0, \ldots, p_{i-1}\}$ be a set of i permissible values. Then, for any subset $S_i \subseteq P_i$, we can define a literal $x_i^{S_i}$ as:

$$x_i^{S_i} = \begin{cases} 1, & \text{if } x_i \in S_i \\ 0, & \text{otherwise} \end{cases}$$

For example, with this extended definition we can write $x_i^{\{0\}}$ to indicate $\bar{x}_i$. It also allows us to write don't care literals as $x_i^{\{0,1\}}$.

A *cube* is a product of of literals, and is 1 if and only if all its literals are 1. Cubes are sometimes called *products* or *product terms*. An EXOR of two cubes evaluates to 1 if and only of the two cubes have different values. An ESOP is the *exclusive-or* (EXOR) of a set of cubes.

It is desirable to be able to use ESOP expressions to represent multiple-output functions. In order to do this, we can use our multiple-valued literal definitions to view cubes as having an *input part* and an *output part*. The input part of a cube is the conjunction of the literals in the cube. The output part of a cube is an additional multiple-valued literal which has as many permissible values as there are outputs. This allows us to share cubes between outputs, which enables the minimization of a single ESOP for multiple outputs, without having to minimize separate ESOPs for each output.

For example, we could write a 3-input 2-output function in the following way:

$$x_1^{\{0\}}x_2^{\{0\}}x_3^{\{1\}}y^{\{0\}} \oplus x_1^{\{0\}}x_2^{\{1\}}x_3^{\{0\}}y^{\{0\}} \oplus \\ x_1^{\{1\}}x_2^{\{0\}}x_3^{\{1\}}y^{\{0,1\}} \oplus x_1^{\{1\}}x_2^{\{0,1\}}x_3^{\{1\}}y^{\{0,1\}}$$

Here, the x_i variables determine the input- and the y variable the output-part of the cubes. In this example, the first two cubes only affect the first output. The last two cubes affect both outputs. Note that variable x_2 is a don't care in the last cube.

We say that cubes C_S and C_R *coincide* in a variable x_i if $x_i^{S_i} \in C_S$, $x_i^{R_i} \in C_R$ and $S_i = R_i$. The *distance* between two cubes is defined to be the number of variables in which the cubes do not coincide. For example, the distance between cubes $x_1^{\{1\}}x_2^{\{0\}}x_3^{\{1\}}y^{\{0,1\}}$ and $x_1^{\{1\}}x_2^{\{0,1\}}x_3^{\{1\}}y^{\{0,1\}}$ is 1.

Over the years, different cube operations have been developed in the context of ESOP minimization. The ExorLink operation is a generalization of several different such transformations [9]. It replaces a pair of cubes by a set of new cubes such that the ESOP function remains unchanged. The formal definition of the ExorLink of two cubes C_S and C_R is as follows [9]:

$$C_S \otimes S_R = \bigoplus \{x_1^{S_1} \cdots x_i^{S_i \oplus R_i} \cdots x_n^{R_n} \mid \forall i : S_i \neq R_i\}$$

The number of cubes that replaces the pair is equal to the distance between the cubes. For example, if we apply ExorLink on two distance-0 cubes, this is equivalent to removing them from the expression.

A *logic network* can be viewed as a directed acyclic graph with some additional structure. Where we would write $G = (V, E)$ to represent a DAG, we write $G = (V_G, E_G, PI_G, PO_G, F_G)$ for a logic network. Here, $PI_G \subseteq V_G$ corresponds to the *primary inputs* to the network, and $PO_G \subseteq V_G$ corresponds to the *primary outputs*. All other nodes $n \in V - (PI_G \cup PO_G)$ correspond to logic gates. Through the logic gates, a logic network computes a multiple-output Boolean function $F_G : \mathbb{B}^{|PI_G|} \to \mathbb{B}^{|PO_G|}$, from the primary inputs to the primary outputs. An *and-inverter graph* (AIG) is a specific type of logic network in which each node corresponds to an AND gate. AIGs may have complemented edges which removes the need for explicit inverter gates.

Given a logic network G and a node $n \in V_G$, a *cut* $C = (n, I, f)$ of G consists of a set of nodes I such that every path from a primary input to n passes through a node in I. The node n is called the *root* or *output* of the cut and the nodes in I are called the *leaves* of the cut. Each cut corresponds to a cut function $f : \mathbb{B}^{|I|} \to \mathbb{B}$, which is a Boolean function defined on the leaves of the cut. Intuitively, one may think of a cut as inducing a subgraph of the logic network that consists of the nodes between the leaves I and the output n. Therefore, cuts can be used to partition a logic network into a set of sub-networks. A *window* $W = (N, I, F)$ is a generalized cut with multiple outputs N. There is no real distinction between logic networks and windows, other than the fact that windows are explicitly subnetworks of some larger network. But we may view a window as a logic network: $W = (V_W, E_W, I, N, F)$, where V_W and E_W are the nodes and edges in the subgraph between the window's leaves and outputs.

III. Algorithm Context

A key requirement of this work was the integration of our algorithm within a larger existing optimization framework. The details of this framework are outside of the scope of this paper, but we give a brief overview to explain the context in which our optimization algorithm takes place. Given a logic network, the framework generates windows of the network. Next, the framework generates truth tables for all nodes within the window. This is done using a fast truth table package that efficiently scales up to 16 variables. Finally, the framework invokes our algorithm on the and replaces it with an optimized one. In other words, given a logic network G, the framework generates a sequence of windows $(G_1, G_2, \ldots, G_n)$ upon which our algorithm acts. Note that this process is completely transparent to our algorithm. It just views these subsequent windows as independent logic networks. Hence, in the remainder we will simply discuss the optimization of logic networks, without referring explicitly to windows.

IV. ESOP Collapsing

The goal of ESOP collapsing is to *flatten* a multi-level logic network to an ESOP expression. This requires us to generate ESOP expressions for all outputs of the network. Since our algorithm operates within a larger framework, the necessary truth tables "for free". Our algorithm uses exploits this by collapsing the truth tables corresponding to output nodes into so-called *Pseudo-Kronecker Forms* (PKRMs). PKRMs are a specific subset of ESOP expressions that can be generated using only positive/negative Davio expansion and Boole's expansion[17]. They are considered to be a good starting point for ESOP minimization [10]. PKRMs are commonly generated from Binary Decision Diagrams (BDDs). However, as our algorithm already has access to truth tables, we prefer not to build additional BDDs for this purpose. Instead, we adapt conventional algorithms and generate PKRMs by using the fast truth table package. We emulate BDDs by using an efficient hash table implementation on top of the truth table package. Our collapsing algorithm then proceeds much like the algorithms presented in [17], [10].

V. ESOP Minimization

After generating the initial seed ESOPs for the network, our goal is now to minimize these expressions. We do so in a manner inspired by the Exorcism family of algorithms [8], [9], [10]. Our minimization algorithm takes as input an initial ESOP cover, from which distance 0/1 cubes have already been eliminated. It then operates on this cover by iteratively applying the distance 2 to 4 ExorLink operation to suitable cube pairs. It iteratively reshapes the cover, accepting ExorLink operations

between pairs of cubes if the result of ExorLinking would reduce the number of cubes or literals.

VI. EXOR Refactoring

After ESOP collapsing and minimization, we have a logic network $G = (V_G, E_G, PI_G, PO_G, F_G)$, and a corresponding minimized multiple-output ESOP expression E. Our goal now is to *factor* this expression in such a way as to find a new multi-level representation for F_G that is more efficient than G (e.g. in terms of logic levels and nodes). Further, we want this representation to be in terms of *primitive gates*. Primitive gates are simple logic gates that might be found in a standard cell library. They typically have a low fanin count and compute a simple logic function. Examples are the AND-2 and EXOR-2 gates which compute the 2-input AND, and 2-input EXOR functions, respectively.

At this point it becomes useful to establish some additional notation. Suppose that ESOP expression E contains m cubes. We say that $\text{cubes}(E) = \bigcup_{i=1}^{m}\{C_i\}$ is the set of cubes contained in E. Let $\text{cubes}(E, i) \subseteq \text{cubes}(E)$ be the subset of cubes that affect output i.

Our refactoring approach separates optimization of the ESOP's EXOR logic from that of it's AND/INV logic, and then merges the two optimized parts back together. As such, our method method consists of two main stages: (i) *cube separation* and (ii) *EXOR-tree construction*.

A. Cube Separation

In this stage we extract the cubes from the EXOR operators and optimize them separately. Recall that cubes are products of literals. Given this, is natural to convert cubes to AIGs. We can then use AIG optimization methods to optimize them to take advantage of logic sharing between cubes.

Suppose that our ESOP expression E contains m cubes. During cube separation we create a new intermediate logic network $G' = (V_{G'}, E_{G'}, I_{G'}, O_{G'}, F_{G'})$. As the ESOP expression has m cubes, G' has m outputs. Therefore, it computes a function $f : \mathbb{B}^{|I|} \rightarrow \mathbb{B}^m$. The function computed by output i is specified by the corresponding cube C_i. In other words, we now have a established a one-to-one mapping $C : \text{cubes}(E) \rightarrow PO_{G'}$.

After construction of G', we decompose it into an AIG network. We then use AIG rewriting techniques to further optimize it. Note that, at this point we have captured the AND/INV logic of E in terms of an optimized, multi-level representation in primitive AND-2 gates. A more detailed description is given in Algorithm 1.

B. EXOR-tree Construction

After finding an efficient representation of the AND/INV portion of the ESOP expression we still need to incorporate the EXOR logic. We do this by building, for each output of E, an balanced EXOR-2 tree on top of the outputs of G'. Note that we can do this because each output of G' corresponds to one of E's cubes. Therefore, every output of original logic network G can be computed simply by applying EXOR operators to the outputs of G'.

By construction, we have $|\text{cubes}(E)| = |PO_{G'}|$. In this second stage we build, for each window output $o_i \in PO_G$, an auxiliary network $G_i = (V_{G_i}, E_{G_i}, PI_{G_i}, PO_{G_i}, F_i)$, such that $PI_{G_i} = C(\text{cubes}(E, i)) \subseteq PO_{G'}$. Recall that C maps cubes to their corresponding outputs in G'. In other words, the inputs to EXOR tree G_i are exactly those outputs of G' that correspond to the cubes used to compute the i-th output of G. Note that F_i is the the i-th projection of F.

Input : Logic network G and minimized ESOP expression E
Output : Refactored logic network G'
Function refactor(G, E)
  set $nout \leftarrow$ num_po(G);
  set $outlists \leftarrow$ new_lists($nout$);
  set $ncubes \leftarrow$ num_cubes(E);
  set $G' \leftarrow$ new_net();
  foreach $i \in \{1, \ldots, ncubes\}$ **do**
    set $C_i \leftarrow$ get_cube(E, i);
    set $PO_{G'_i} \leftarrow$ net_add_output(G', C_i, i);
    foreach $j \in \{1, \ldots, nout\}$ **do**
      if (is_output_affected(C_i, j) **then**
        list_add($outlists[j], PO_{G'_i}$) ;
    end
  end
  aig_decompose(G') ;
  aig_optimize(G') ;
  foreach $i \in \{1, \ldots, nout\}$ **do**
    set $G_i \leftarrow$ balanced_exor_decomp($outlists[i]$);
    net_add_output($G', G_i, ncubes + i$)
  end
  foreach $i \in \{1, \ldots, ncubes\}$ **do**
    net_del_output(G', i)
  end
return G';

Algorithm 1: Our multi-stage refactoring algorithm

In order to build these EXOR trees, we keep track of which cubes of E belong to which outputs. This allows us to incrementally build up the sets $\text{cubes}(E, i)$ for each output. As these cubes correspond to outputs of G', we can build a balanced EXOR tree by treating the outputs $PO_{G'}$ of G' as the leaves of the tree G_i. Such a tree can be computed recursively, starting from the initial set $\text{cubes}(E, i)$. By removing the cube outputs from G' and adding to it the EXOR tree outputs PO_{G_i}, we obtain a network whose functionality is by construction equivalent to that of G. This merged network has only primitive AND/INV gates at the bottom and primitive EXOR gates at the top. Implementation details can be found in Algorithm 1.

VII. Experiments

Our first experiment attempts to validate the hypothesis that ESOP minimization can unlock optimizations that are inaccessible to SOP based refactoring. In our second experiment, we integrate our algorithm into a commercial synthesis tool, and evaluate its added value after physical synthesis.

A. SOP vs. ESOP Refactoring

In this experiment we contrast our ESOP optimization algorithm with an analogous SOP optimization algorithm. The SOP algorithm first performs SOP collapsing, then SOP minimization, and finally refactors the network by decomposing the SOPs into primitive AND-2/OR-2 gates.

Table I shows the a comparison between the two algorithms, on a set of EXOR-heavy benchmarks. Looking at gate count and logic levels, the results confirm our hypothesis. For EXOR-heavy logic, our ESOP optimization method is able to discover EXOR gates and effectively use them to refactor networks more efficiently. This is especially in terms of logic levels. For all benchmarks, our refactoring method reduces the number of logic levels, as compared to the SOP based one. The reduction is significant and ranges up to 83.3% for the parity benchmark and 51.7% for the alu4 benchmark. On average, we reduce the number of levels by 44.6%.

TABLE I. A COMPARISON OF SOP REFACTORING AND ESOP REFACTORING ON A SET OF EXOR-HEAVY BENCHMARKS.

		SOP					ESOP				
Benchmark	I/O	Gates	Terms	Literals	Levels	Runtime (s)	Gates	Terms	Literals	Levels	Runtime (s)
rd53	5/3	48	48	93	9	0.0	32	47	94	7	0.0
rd73	7/3	120	120	237	14	0.0	122	172	344	9	0.0
rd84	8/4	116	116	229	11	0.1	205	271	542	10	0.1
parity	16/1	46	46	91	31	6.0	15	30	60	5	0.0
9sym	9/1	194	194	387	14	0.0	222	271	542	11	0.1
sym10	10/1	308	308	615	22	0.1	395	436	872	12	0.1
misex3	14/14	3850	3850	7688	26	1.0	2854	4039	8078	14	25.1
alu4	14/8	1983	1983	3961	29	1.3	1386	1745	3490	14	3.4
f51m	8/8	116	116	225	12	0.0	99	145	289	8	0.0
b12	15/9	75	75	146	9	0.0	95	124	147	8	0.0
Average		685.6	685.6	1367.2	17.7	0.85	**538.9**	728	1445.8	**9.8**	2.88

TABLE II. POST PLACE&ROUTE RESULTS ON 46 INDUSTRIAL DESIGNS

Flow	Area	Leakage	WNS	TNS	Runtime
Baseline	1	1	1	1	1
Proposed flow	**-1.08%**	**-0.92%**	**-0.15%**	+0.86%	+3.93%

For 5 out of 10 benchmarks, our method is also to reduce the number of gates. Again, this reduction can be quite significant, ranging up to 67.4% for the parity benchmark, and 25.9% for the misex3 benchmark. On average, our method reduces the number of gates by 21.4%.

Finally, Table I shows that in terms of runtime our ESOP based method is competitive with the SOP based one. The only exception to this rule is the misex3 benchmark. Runtime in this benchmark is dominated by the ESOP minimization phase of our algorithm. Our ESOP minimization finds many beneficial ExorLink operations. Hence, it performs many iterations. Although it is easy to adapt our algorithm to trade-off runtime for quality-of-results (e.g. by limiting the number of iterations), in this experiment we are interested examining the best results achievable by our method.

B. Integration & Industrial Benchmarks

For our second experiment test the added value of our algorithm on set of 46 industrial benchmarks. We add our algorithm as an optimization script to a commercial tool, augmenting its existing set of deep logic optimization scripts. We then use the commercial tool to perform its full synthesis flow, which encompasses RTL optimization, logic synthesis, and physical synthesis, including placement and routing. We compare our results to a baseline run of the tool and find that our method unlocks optimizations that are out of the tool's baseline optimizations. Table II sums up the results. It shows that the logic optimizations found by our method carry over in the optimization pipeline after place&route. Our method improves area by 1.08%, leakage power by 0.98%, and worst negative slack by 0.15%, at some runtime cost.

Note that, for an industrial logic optimization algorithm, a 1% reduction in area after physical synthesis is generally considered significant [18]. It shows that the gains achieved in the logic optimization phase carry over, and that they are not absorbed by existing logic- or physical optimization algorithms.

VIII. CONCLUSIONS

We present a logic refactoring algorithm based on the two-level ESOP representation. Theoretical results predict the compactness of ESOPs as compared to SOPs, for important classes of functions. Our algorithm shows that we can use this compactness to our advantage. For EXOR-heavy logic, our algorithm unlocks optimization opportunities that are not reached by similar approaches based on SOP refactoring. Our algorithm dominates SOP collapsing in terms of logic levels for each network in the tested benchmarks. It also significantly decreases the average number of gates. We have integrated our algorithm in a commercial synthesis tool. By testing the augmented tool on a set of industrial benchmarks, we find that the inclusion of our algorithm enables additional optimizations in area, power, and slack, after physical synthesis.

REFERENCES

[1] T. Sasao, "EXMIN2: A Simplification Algorithm for Exclusive-OR-Sum-of Products Expressions for Multiple-Valued-Input Two-Valued-Output Functions," *IEEE TCAD*, vol. 12, no. 5, pp. 621–632, 1993.

[2] M. A. Perkowski and M. Chrzanowska-Jeske, "An Exact Algorithm to Minimize Mixed-Radix Exclusive Sums of Products for Incompletely Specified Boolean Functions," in *Proc. ISCAS*, 1990, pp. 1652–1655.

[3] T. Sasao, "An Exact Minimization of AND-EXOR Expressions Using Reduced Covering Functions," in *Proc. SSMII*, 1993, pp. 374–383.

[4] A. Gaidukov, "Algorithm to derive minimum ESOP for 6-variable function," in *5th IWBP*, 2006.

[5] M. Helliwell and M. A. Perkowski, "A Fast Algorithm to Minimize Multi-Output Mixed-Polarity Generalized Reed-Muller Forms," in *Proc. IEEE ISCS*, pp. 17–20.

[6] M. A. Perkowski, M. Helliwell, and P. Wu, "Minimization of Multiple-valued Input Multi-Output Mixed- Radix Exclusive Sums of Products for Incompletely Specified Boolean Functions," in *Proc. ISMVL*, May 1989, pp. 256–263.

[7] T. Sasao, "Exmin: A simplification algorithm for exclusive-or-sum-of-products expressions for multiple-valued input two-valued output functions," in *Proc. ISMVL*, 1990, pp. 128–135.

[8] N. Song and M. A. Perkowski, "EXORCISM-MV-2 : Minimization of Exclusive Sum of Products Expressions for Multiple-valued Input incompletely Specified Functions," in *Proc. ISMVL*, 1993, pp. 132–137.

[9] ——, "Minimization of Exclusive Sum of Products Expressions for Multiple-Valued Input," *IEEE TCAD*, vol. 15, no. 4, pp. 385–395, 1996.

[10] A. Mishchenko and M. A. Perkowski, "Fast heuristic minimization of exclusive-sums-of-products," in *Proc. RM Workshop*, 2001, pp. 242–250.

[11] T. Sasao, *Logic Synthesis and Optimization*, 1st ed. Kluwer Academic Publishers, 1993.

[12] G. De Micheli, *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, 1994.

[13] F. Luccio and L. Pagli, "On a New Boolean Function with Applications," *IEEE Transactions on Computers*, vol. 48, no. 3, pp. 296–310, 1999.

[14] A. Bernasconi, V. Ciriani, R. Drechsler, and T. Villa, "Logic Minimization and Testability of 2SPP Networks," *IEEE TCAD*, vol. 27, no. 7, pp. 1190–1202, 2008.

[15] V. Ciriani, "Synthesis of SPP Three-Level Logic Networks Using Affine Spaces," vol. 22, no. 10, pp. 1310–1323, 2003.

[16] B. Steinbach and C. Posthoff, "Compact XOR Bi-Decomposition for Generalized Lattices of Boolean Functions," in *Proc. RMW*, 2017.

[17] R. Drechsler, "Pseudo-Kronecker Expressions for Symmetric Functions," *IEEE TCAD*, vol. 48, no. 9, pp. 987–990, 1999.

[18] L. Amarù, P. Vuillod, J. Luo, and J. Olson, "Logic Optimization and Synthesis: Trends and Directions in Industry," in *DATE*, 2017.