2017 IEEE 47th International Symposium on Multiple-Valued Logic

Classifying Functions with Exact Synthesis

Winston Haaswijk Eleonora Testa Mathias Soeken Giovanni De Micheli
Integrated Systems Laboratory, EPFL, Lausanne, VD, Switzerland

***Abstract*—Due to recent advances, constraint solvers have become efficient tools for synthesizing optimum Boolean circuits. We take advantage of this by showing how SAT based exact synthesis may be used as a method for finding minimum length Boolean chains. As opposed to other exact synthesis methods, ours may be easily parallelized, which we use to obtain a speedup of approximately 48 times. By combining our method with NPN canonization, we find for the first time the minimum length chains for all 4- and 5-input functions in terms of 3-input Boolean operators. Finally, we propose a hardware acceleration method for NPN canonization. It can be used to speed up NPN canonization in existing algorithms, and we believe it will allow us to find all 6-input NPN classes as well.**

I. Introduction

Every n-input Boolean function f over inputs $x_1, \ldots, x_n$ can be expressed in terms of a Boolean chain. Such a chain is a sequence of steps that combine previous steps or inputs using Boolean operators from a set of primitives P. In the context of digital design it is typical to refer to a Boolean chain and steps as a *logic network* and *gates*, respectively. But as we are considering some fundamental properties of Boolean functions, we avoid using such hardware jargon. The minimum length of a Boolean chain to compute f is referred to as *combinational complexity* of f, and denoted $C(f)$ [1]. More formally, a Boolean chain for corresponding to function of n inputs is a sequence $(x_{n+1}, \ldots, x_{n+r})$, where

$$x_i = g_i(x_{1(i)}, x_{2(i)}, \ldots, x_{m(i)}) \quad \text{for } n+1 \leq i \leq n+r.$$

In other words, each step in the chain combines m previous steps or inputs with $x_{1(i)} < x_{2(i)} < \cdots < x_{m(i)} < x_i$ using the m-input Boolean operator g_i. Note that this definition allows for multiple *fanouts*: multiple distinct steps in the chain may refer to the same input or step x_j.

There are several research directions that address questions about combinational complexity in different ways. We may consider these directions as consisting of three different categories. The first category is concerned with finding set of primitives such that the complexity of all Boolean functions f satisfies some upper bound (see, e.g., [2]–[4]). The second is concerned with finding complex Boolean functions that satisfy a lower bound for some given set of primitives (see, e.g., [5]–[7]). Finally, the third is concerned with finding exact numbers for the combinational complexity given a subset of Boolean functions and a set of primitives P (see, e.g., [1], [8], [9]).

The work we present in this paper falls into the third category. Having exact numbers for the combinational complexity of some small functions can help to find tighter upper bounds for larger functions by using arguments from Boolean decomposition. Further, efficient methods for finding small Boolean chains has applications in logic synthesis and optimization. For example, logic rewriting algorithms optimize logic networks by replacing subnetworks by optimized Boolean chains [8], [10].

TABLE I
Combinational complexity of all 4-input functions using 2-input operators [1]

$C(f)$	Classes	Functions
0	2	10
1	2	60
2	5	456
3	20	2474
4	34	10624
5	75	24184
6	72	25008
7	12	2720

TABLE II
Combinational complexity of all 5-input functions using 2-input operators [1]

$C(f)$	Classes	Functions
0	2	12
1	2	100
2	5	1140
3	20	11570
4	93	109826
5	389	995240
6	1988	8430800
7	11382	63401728
8	60713	383877392
9	221541	1519125536
10	293455	2123645248
11	26535	195366784
12	1	1920

Knuth [1] has computed the combinational complexity of all 4- and 5-input Boolean functions composed of all 2-input Boolean operators. In this work we repeat the experiment by using all 3-input Boolean operators as set of primitives. Following Knuth we make use of the property that all functions that are equivalent up to input negation, input permutation, and output negation (NPN equivalence, [11]) have the same combinational complexity. This allows us to consider a subset of 222 and 616,126 functions instead of 65,536 and 4,294,967,296 functions for 4 and 5 inputs, respectively.

In fact, the exact numbers for the combinational complexity for all 222 NPN classes of the 4-input functions can be found efficiently by enumerating all Boolean chains until some chain for each function has been encountered [1]. Table I lists the combinational complexity for all 4-input functions. A more

2378-2226/17 $31.00 © 2017 IEEE
DOI 10.1109/ISMVL.2017.44

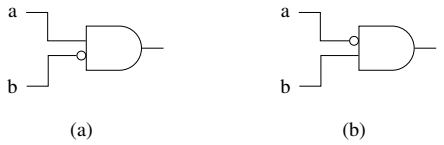

Fig. 1. An example of two different functions that are P-equivalent. The circuit in (a) computes $f = a \cdot \bar{b}$ and the circuit in (b) computes $g = \bar{a} \cdot b$. They can be made equivalent by swapping the inputs to the AND gate.

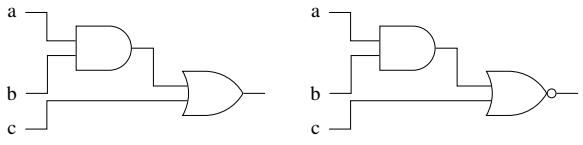

Fig. 2. An example of two different functions that are NPN-equivalent. The circuit in (a) computes $h = a \cdot b + c$ and the circuit in (b) computes $i = \bar{a} \cdot \bar{c} + \bar{b} \cdot \bar{c}$. They are NPN-equivalent because circuit (a) can be made equivalent to circuit (b) by inverting its output.

sophisticated algorithm is required to find exact numbers for the combinational complexity for all 616,126 NPN classes of the 5-input functions. Yet, "thanks to a bit of good luck" (as stated in [1]) and the computer program `BOOLCHAINS`,[1] it was possible to find the numbers presented in Table II. However, modifications to the main algorithm were required to find the numbers for some of these classes. Certain classes were handled as special cases. The vast majority of computation time was spent finding the 11-step chains for their 6 corresponding NPN classes and the 12-step chain for its single corresponding NPN class [1].

We propose the use of SAT based exact synthesis to find the combinational complexity of all 4- and 5-input functions, and show that it is an efficient method for doing so. Exact synthesis [12]–[14] is a method that, given a set of primitives, finds optimum Boolean chains. We adjust the typical exact synthesis formulation by considering 3-input Boolean operators as the set of primitives. Our findings are that exact synthesis can efficiently find optimum Boolean chains for all 4- and 5-input NPN classes. It does so without the requirement to explicitly enumerating all Boolean chains or to use any case distinctions. Further, our method does not require any modifications to handle special cases. Finally, our method is easily parallelized.

II. Preliminaries

A. NPN Canonization

Two functions are P-equivalent if they are equivalent up to permutation of their inputs. For example the functions $f = a \cdot \bar{b}$ and $g = \bar{a} \cdot b$ are P-equivalent, since we can make them equal by swapping the inputs a and b (see Figure 1). NPN equivalence is a generalization of P equivalence. We say that two functions are NPN-equivalent if they are equivalent up to permutation of their inputs *and* negation of their inputs and output [15], [16]. For example, the functions $h = a \cdot b + c$ and $i = \bar{a} \cdot \bar{c} + \bar{b} \cdot \bar{c}$ are NPN-equivalent, since h can be made equivalent to i by negating its output (see Figure 2).

The number of single output n-input functions is 2^{2^n}. This number grows rapidly as we increase n. NPN canonization is a way to group Boolean functions into classes that are equivalent up to permutation and negation of inputs and output. Due to this grouping into classes we use the terms NPN canonization and NPN classification interchangeably. We can view an NPN equivalence class as a set of functions $[f]$. When two functions f and g are part of the same NPN class (i.e., $f \in [f]$ and $g \in [f]$), we say that they are NPN-equivalent. We pick one function $\hat{f} \in [f]$ to be the *equivalence class representative*. We say that $\hat{f}$ is the *canonical* representative of $[f]$. Typically, the function $f \in [f]$ whose truth table corresponds the smallest integer value is chosen to be the equivalence class representative.

[1] http://www-cs-faculty.stanford.edu/~uno/programs/boolchains.tgz

TABLE III
Comparing the number of n-input functions and NPN classes

n	Number of functions	Number of NPN classes
0	1	1
1	4	2
2	16	4
3	256	14
4	65,536	222
5	4,294,967,296	616,126
6	18,446,744,073,709,551,616	200,253,952,527,184

In general, Boolean function classification is useful when we want to learn something about all functions. For example, in this paper we are interested in finding the minimum length Boolean chain for all 5-input functions. Since NPN equivalence relies only on permutations and negations, it does not affect the length of Boolean chains. The minimum length Boolean chain for any $f \in [f]$ can be derived from chain for $\hat{f}$, simply by applying the proper permutations and negations. More formally, $C(f) = C(g)$ if $g \in [f]$. Hence, to find $C(f)$ for all n-input functions we do not need to examine all functions. Instead we can find $C(\hat{f})$ only for the NPN class representatives. This is preferable, since the number of NPN classes is significantly smaller than the number of functions. Table III lists the number of functions and classes for up to 6 inputs to illustrate how significant this difference is. NPN classification has applications ranging from Boolean matching to logic rewriting and exact synthesis [17]–[19]. Efficient exact and heuristic algorithms for NPN classification have been developed over the years [17], [20], [21].

B. Exact Synthesis

Given a set of primitives P, an exact synthesis algorithm finds the optimum representation for a Boolean function in terms of P. Given different objectives, different representations may be considered optimum. For example, we may be looking for the representation with the smallest depth, or for the smallest possible Boolean chain. For example, suppose that P consists only of the 2-input NAND operator. We could then use exact

synthesis to find the smallest possible chain of NAND operators that implements a given function.

Exact synthesis is a special case of the well-known minimum circuit size problem (MCSP) [22]. No polynomial time algorithm for either MCSP or exact synthesis is known, and they are conjectured to be computationally intractable [23]. Due to its complexity, the application of exact synthesis has been limited to small functions, i.e., functions of 4 or 5 inputs.

There are various ways in which we can implement an exact synthesis algorithm [8], [13], [24]. Recently, a class of implementations that view exact synthesis as a SAT/SMT problem has been getting more attention. We describe how the problem may be formulated in such a way that it can be solved by a SAT/SMT solver. Suppose that we are given the set of primitives P and the Boolean function f. Our goal is to find the smallest possible Boolean chain that computes f in terms of P. Let us assume that our constraint solver has a procedure $\mathrm{solution\_exists}(f, g)$ which returns true if and only if there exists a Boolean chain of g operators from P that computes f. Our goal then reduces to finding the minimum g for which this is true. We start by setting $g \leftarrow 0$, and checking if $\mathrm{solution\_exists}(f, g)$ is true. If not, we increment g and try again until we find a value for g that works. The first g for which $\mathrm{solution\_exists}(f, g)$ returns SAT is the minimum. The procedure $\mathrm{solution\_exists}(f, g)$ can be expressed as a SAT or SMT formula. Hence, we can solve the exact synthesis problem with a SAT or SMT solver. Other optimization criteria, such finding the Boolean chain with minimum depth, can be achieved similarly.

III. Exact Synthesis Method

Our goal in this paper is to find the minimum length Boolean chains for all 4- and 5-input functions in terms of 3-input Boolean operators. In other words, P consists of all 3-input Boolean functions. We present a method that achieves this goal by using NPN classification and exact synthesis. Roughly, our method can be divided into two parts: (i) finding all 5-input NPN classes, and (ii) using exact synthesis to find all minimum length Boolean chains.

A. Finding All NPN Classes

In order to find the representative $\hat{f}$ for a given function f, one needs to visit all functions in $[f]$ to find the smallest one. If f has no helpful properties—such as symmetries in the inputs (see, e.g., [20], [25])—one needs to apply all possible combinations of 2^n input negations and $n!$ input permutations for both f and $\bar{f}$. In order to reduce the effort, we can use a smart ordering in which all these transformations are applied. We can use gray code enumeration to invert inputs, thereby flipping only one bit at a time. In a similar way we can use *plain changes* (see, e.g., [26]) to visit all permutations by swapping two adjacent inputs at each time. Note that it is possible to combine both concept in an enumeration algorithm that visits all *signed permutations*, i.e., permutations in which elements can be complemented [1].

Since all elements in $[f]$ are visited when computing $\hat{f}$, we can store this information while enumerating all representatives of all NPN classes. In the case $n = 5$, which we consider, there are $2^{2^5} = 4,294,967,296$ single output 5-input functions. We initialize a map R that is indexed by 5-input functions and initialize each of its 2^{2^5} elements with Λ (null). In a loop we find some f for which $R(f) = \Lambda$, and compute $\hat{f}$ using the algorithm described above. While visiting all elements f' in $[f]$ we set $R(f) \leftarrow \hat{f}$. Obviously, this loop needs to be traversed only 616,126 times, the number of NPN classes for 5-input functions. Afterwards, the image of R is the set of all representatives.

B. Finding Minimum Length Chains With Exact Synthesis

After finding the NPN classes, we use exact synthesis to find the minimum length Boolean chains. We use a SAT based method similar to the one introduced in Section II-B. Our algorithm is an extension of the SAT formulation proposed by Knuth in [12]. This formulation is itself based on earlier work from Kojevinok et al. [13] and Eén [14]. To find all minimum length chains, we simply apply our exact synthesis to every 4- and 5-input NPN class.

A key difference between our synthesis algorithm and previous approaches is that our set of primitives P consists of all 3-input Boolean operators. By contrast, in earlier approaches P typically consist of 2-input operators, or a small subset of 2- and 3-input operators [8], [9], [12]. The use of 3-input operators significantly speeds up synthesis time. We are not aware of any other exact synthesis algorithm that is able to find these minimum length Boolean chains, at least not without handling special cases differently. Moreover, using 3-input operators leads to novel results: the minimum length chains in terms of 2-input operators are known, but the minimum chains in terms of 3-input operators are not.

Another difference between our method and others is that ours is easily parallelized. Exact synthesis may be invoked in parallel on every NPN class we find, as there are no dependencies between invocations. Other methods would be significantly harder to parallelize. For example, enumeration based methods work by doing a search of different circuit structures [1]. This search proceeds sequentially, yielding a sequence of optimum chains. This is not a process that is trivial to parallelize, as lower parts of the search space tree depend on choices made above.

For both 4-input and 5-input functions we can find tight upper bounds on the length of minimum Boolean chains. Our set of primitives P includes the 2-to-1 multiplexer, as it has a corresponding 3-input Boolean operator. We can use this operator to efficiently decompose functions and to find an upper bound. For example, we can write any 4-input Boolean function as $f(x_1, x_2, x_3, x_4) = \bar{x}_1 \cdot f(0, x_2, x_3, x_4) + x_1 \cdot f(1, x_2, x_3, x_4)$. This is known as Boole's expansion, and can be implemented by a 2-to-1 multiplexer. Note that the cofactors of f are 3-input functions. This means that they can both be implemented by a single operator from P. Figure 3 shows a sketch of this decomposition. Therefore, by using a multiplexer to do the

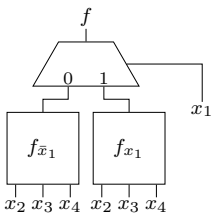

Fig. 3. We can implement any 4-input Boolean function by using at most three 3-input operators.

TABLE IV
COMBINATIONAL COMPLEXITY OF ALL 4-INPUT FUNCTIONS USING 3-INPUT OPERATORS

$C(f)$	Classes	Functions	Computation time (sec) Avg.	Max.	Total
0	2	10	0.000	0.000	0.000
1	12	932	0.001	0.002	0.014
2	117	34,250	0.001	0.002	0.173
3	91	30,344	0.003	0.005	0.245

initial expansion and two operators to implement the cofactors, we can implement any 4-input function with at most 3 operators from P. A similar argument can be used to show that we can implement any 5-input function using at most 7 operators.

C. Experimental Results

Using NPN classification reduces the number exact invocations. However, at 616,126 5-input classes the number is still large. To make the run time of our experiment practical we run exact synthesis in parallel. We use a machine with 2 Intel Xeon E5-2680 v3 (Haswell) CPUs, each of which has 12 cores with support for 2 hyperthreads. Since the problem is embarrassingly parallel, we take full advantage of our hardware by using 48 threads.

We first use our method to find the minimum length Boolean chains for all 4-input operators. The results of this can be found in Table IV. This table shows the number of NPN classes that have a specific combinational complexity $C(f)$, as well as the corresponding number of functions with that complexity. Further, it shows run time information. For each value of $C(f)$ we show the average, maximum, and total synthesis run time (in seconds) for all the NPN classes with that combinational complexity. The results show that our synthesis algorithm is efficient: it never requires more than 0.005 seconds to synthesize any 4-input function.

The results also show that the upper bound we derived for 4-input functions in Section III-B is exact. There are exactly 91 NPN classes, corresponding to 30344 functions, that cannot be implemented by using fewer than 3 operators.

Next, we apply our method to finding the minimum length chains for all 5-input functions. The results can be found in Table V. Interestingly, the upper bound we found in Section III-B was not exact in this case. Any 5-input function can be implemented by using at most 5 3-input operators.

TABLE V
COMBINATIONAL COMPLEXITY OF ALL 5-INPUT FUNCTIONS USING 3-INPUT OPERATORS

$C(f)$	Classes	Functions	Computation time (sec) Avg.	Max.	Total
0	2	12	0.000	0.000	0.000
1	12	2,280	0.001	0.002	0.017
2	311	395,676	0.003	0.005	0.911
3	12,257	58,519,472	0.021	0.089	260.550
4	339,739	2,321,397,216	1.805	57.898	613,082.000
5	263,805	1,914,652,640	18.550	3,261.770	4,893,600.000

As shown by Table V, exact synthesis turns out to be an efficient method for finding the minimum chains. No function requires more than an hour to be synthesized, and the average synthesis time is just 8.938 seconds. This means that our method is able to find all minimum length chains, without having to resort to different handling of special cases.

Despite the efficiency of our method, the total sequential CPU time necessary to find *all* minimum length chains is still $8.938 \times 616,126 = 5,506,943.478$ seconds, simply due to the large number of functions. However, one useful property of our method is that it can be easily parallelized. By running it on 48 threads in parallel the total wall clock time reduces significantly. We are able to synthesize all functions in $\frac{5,506,943.478}{48} = 114727.99$ seconds. As there are 86,400 seconds in a day, this allows us to complete synthesis in just $\frac{114,727.99}{86,400} = 1.3$ days. In other words, we are able to take full advantage of the embarrassingly parallel nature of the problem, and to obtain a speedup of approximately 48 times. Other exact synthesis methods, such as those based on exhaustive enumeration of Boolean chains, are much harder to parallelize. Therefore, even if we suppose they have better run time per function, they may still not be as practical as our method.

IV. CONCLUSIONS AND OUTLOOK

We present a method for finding the minimum length Boolean chains for all 4- and 5-input functions in terms of 3-input operators. Our method is based on NPN classification and exact synthesis. It can also be easily parallelized, enabling an approximate $48\times$ reduction in run time. We show that our exact synthesis implementation is efficient and able to find all minimum length chains without needing to handling any special cases differently. For the first time, we present the lengths of these minimum chains as well as their implementations.

Besides being of academic interest, finding minimum chains has practical application. For example, they can be used in logic optimization and technology mapping. It is also motivated by emerging and existing technologies. In recent years different nanotechnologies have been implementing more powerful devices that go beyond the capabilities of traditional NAND/NOR gates [27]–[29]. These devices implement more expressive operators, such as 3-input majority or minority functions. More traditionally, gates such as multiplexers also correspond to 3-input operators. Hence, finding optimum chains

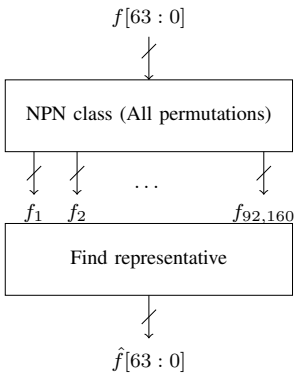

Fig. 4. NPN classification circuit

based on 3-input operators can help in the design of circuits based on these technologies.

Classifying 6-input functions—an outlook: The classification results for 4- and 5-input functions are a self-contained result and very helpful both in theory and practice. But what if we'd like to go a step further and investigate the combinational complexity of all 6-input functions? There exist more than 200 trillion NPN classes and to compute them requires a lot of time making our classification approach infeasible. We suggest to investigate whether hardware acceleration can be a solution to overcome this limitation and compute NPN classes using dedicated hardware. Such hardware can exploit full parallelism in computing the NPN representative as shown by the design in Fig. 4. Input to the circuit is a function f and output its representative $\hat{f}$. Both functions are represented as 64-bits vectors. The circuit consists of one block to enumerate all NPN classes and one block to calculate the representative given all elements in the class.

Input to the first block is the 6-input function f. This block considers all $2^6 \cdot 6! = 46,080$ permutations for f, given by permuting and negating inputs. At the circuit level, both these operations on inputs result in considering different orders for the 64 bits of f. The same $46,080$ permutations are considered for $\bar{f}$. Hence, the first block has $92,160$ 64-bit outputs representing all functions in $[f]$. The second block finds the minimum of all these functions by a balanced tree of comparators, resulting in $\lceil \log_2 92,160 \rceil = 17$ levels. The output of this comparator tree is the smallest functions, i.e., the representative $\hat{f}$.

ACKNOWLEDGMENTS

This research was supported by H2020-ERC-2014-ADG 669354 CyberCare and the Swiss National Science Foundation (200021-169084 MAJesty).

REFERENCES

[1] D. E. Knuth, *The Art of Computer Programming*. Upper Saddle River, New Jersey: Addison-Wesley, 2011, vol. 4A.

[2] M. L. Furst, J. B. Saxe, and M. Sipser, "Parity, circuits, and the polynomial-time hierarchy," *Mathematical Systems Theory*, vol. 17, no. 1, pp. 13–27, 1984.

[3] J. Håstad, "Almost optimal lower bounds for small depth circuits," in *Annual ACM Symposium on Theory of Computing*, 1986, pp. 6–20.

[4] W. Hesse, E. Allender, and D. A. M. Barrington, "Uniform constant-depth threshold circuits for division and iterated multiplication," *J. Comput. Syst. Sci.*, vol. 65, no. 4, pp. 695–716, 2002.

[5] N. Blum, "A Boolean function requiring $3n$ network size," *Theor. Comput. Sci.*, vol. 28, pp. 337–345, 1984.

[6] W. J. Paul, "A $2.5n$-lower bound on the combinational complexity of Boolean functions," *SIAM J. Comput.*, vol. 6, no. 3, pp. 427–443, 1977.

[7] C. Schnorr, "The combinational complexity of equivalence," *Theor. Comput. Sci.*, vol. 1, no. 4, pp. 289–295, 1976.

[8] M. Soeken, L. G. Amarù, P. Gaillardon, and G. De Micheli, "Optimizing majority-inverter graphs with functional hashing," in *Design, Automation and Test in Europe*, 2016, pp. 1030–1035.

[9] R. Schroeppel, "A few mathematical experiments," talk at Experimental Mathematics Workshop, slides at http://richard.schroeppel.name:8015/expmath04-schroeppel-talk.pdf.

[10] W. Haaswijk, M. Soeken, L. Amarú, P.-E. Gaillardon, and G. De Micheli, "A Novel Basis for Logic Rewriting," in *ASPDAC*, 2017.

[11] E. Goto and H. Takahasi, "Some theorems useful in threshold logic for enumerating Boolean functions," in *International Federation for Information Processing Congress*, 1962, pp. 747–752.

[12] D. E. Knuth, *The Art of Computer Programming, Volume 4, Fascicle 6: Satisfiability*. Reading, Massachusetts: Addison-Wesley, 2015.

[13] A. Kojevnikov, A. S. Kulikov, and G. Yaroslavtsev, "Finding efficient circuits using sat-solvers," in *Theory and Applications of Satisfiability Testing*, 2009, pp. 32–44.

[14] N. Eén, "Practical SAT - a tutorial on applied satisfiability solving," in *FMCAD*, 2007.

[15] M. Harrison, "The Number of Equivalence Classes of Boolean Functions Under Groups Containing Negation," *Electronic Computers, IEEE Transactions on*, vol. 12, pp. 559–561, 1963.

[16] S. L. Hurst, D. M. Miller, and J. C. Muzio, *Spectral Techniques in Digital Logic*. Academic Press, 1985.

[17] Z. Huang, L. Wang, Y. Nasikovskiy, and A. Mishchenko, "Fast Boolean matching based on NPN classification," in *Int'l Conf. on Field-Programmable Technology*, 2013, pp. 310–313.

[18] A. Mishchenko, S. Chatterjee, and R. K. Brayton, "DAG-aware AIG rewriting a fresh look at combinational logic synthesis," in *Design Automation Conference*, 2006, pp. 532–535.

[19] W. Haaswijk, M. Soeken, L. Amarú, P.-E. Gaillardon, and G. De Micheli, "LUT Mapping and Optimization for Majority-Inverter Graphs," in *IWLS*, 2016.

[20] A. Petkovska, M. Soeken, G. De Micheli, P. Ienne, and A. Mishchenko, "Fast Hierarchical NPN Classification," in *Field Programmable Logic and Applications*, 2016, pp. 1–4.

[21] M. Soeken, A. Mishchenko, A. Petkovska, B. Sterin, P. Ienne, R. K. Brayton, and G. De Micheli, "Heuristic NPN Classification for Large Functions Using AIGs and LEXSAT Mathias," in *International Conference on Theory and Applications of Satisfiability Testing*, 2016, pp. 212–227.

[22] V. Kabanets and J.-Y. Cai, "Circuit minimization problem," *Proceedings of the thirty-second annual ACM symposium on Theory of computing - STOC '00*, pp. 73–79, 2000.

[23] C. D. Murray and R. R. Williams, "On the (non) np-hardness of computing circuit complexity," in *Conference on Computational Complexity*, vol. 30, 2015, pp. 365–380.

[24] E. Lawler, "An approach to multilevel boolean minimization," *Journal of the ACM*, vol. 11, no. 3, pp. 283–295, 1964.

[25] A. Abdollahi and M. Pedram, "A new canonical form for fast Boolean matching in logic synthesis and verification," in *Design Automation Conference*, 2005, pp. 379–384.

[26] E. Morris, *The history and art of change ringing*. Chapman & Hall, 1931.

[27] P. D. Tougaw and C. S. Lent, "Logic devices implemented using quantum cellular automata," *Journal of Applied Physics*, vol. 75, no. 3, 1994.

[28] D. Lee, W. S. Lee, C. Chen, F. Fallah, J. Provine, S. Chong, J. Watkins, R. T. Howe, H. S. P. Wong, and S. Mitra, "Combinational logic design using six-terminal nem relays," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 32, no. 5, pp. 653–666, 2013.

[29] M. De Marchi, D. Sacchetto, J. Zhang, S. Frache, P.-E. Gaillardon, Y. Leblebici, and G. De Micheli, "Top–down fabrication of gate-all-around vertically stacked silicon nanowire fets with controllable polarity," *IEEE Transactions on Nanotechnology*, vol. 13, no. 6, pp. 1029–1038, 2014.