Are XORs in logic synthesis really necessary?

Ino Háleček, Petr Fišer, Jan Schmidt
Faculty of Information Technology
Czech Technical University in Prague
Prague, Czech Republic
Email: {halecivo, fiserp, schmidt}@fit.cvut.cz

***Abstract*—This paper follows recent research on insufficient synthesis performance for XOR-intensive circuits, and introduces a novel logic representation with a native support of XOR gates, the XOR-AND-Inverter Graphs (XAIG). A rewriting algorithm over XAIG has been implemented in the logic synthesis and optimization package ABC, as the first step towards a complete synthesis process. The results show that XAIG based rewriting can help to discover XORs and improves the area of a mapped network in some cases.**

I. Introduction

Even though the process of logic synthesis and optimization seemed to be an already efficiently resolved problem in past decades, recently there appeared several brand new approaches to it, mostly based on novel data structures used for representing functions and networks.

Originally, when the very first logic synthesis EDA tools emerged, the ultimate function representation was a sum-of-products (SOP) [1], [2]. Two-level SOP minimizers have been proposed as tools [2], [3], [4]. Subsequently, multi-level optimization algorithms and tools based on the SOP representation of network nodes have been devised [5], [6], [7], [8].

As a simpler alternative to general SOP nodes, network representation based on NOR gates has been used in many algorithms [9]. However, such a representation just offered a simple way of algorithms implementation, without having any significant impact to theory.

A big breakthrough in a function representation was the introduction of Binary Decision Diagrams (BDDs) [10], [11]. Synthesis and optimization algorithms were adapted to this structure, making them perform much better [12], [13], [14], [15].

Even though these representations have been used for many decades both in academic and professional synthesis tools [7], [8], they suffer from one problem: they are not scalable. Even though a network of any size can be constructed of SOP nodes, the scalability of nodes is limited. Also the size of BDDs may grow exponentially with the number of inputs, which is unfortunately the case of many contemporary designs.

For this reason, a very efficient network representation, the And-Inverter-Graph (AIG), has been proposed in 2000's [16], [17], [18]. Here a Boolean network (circuit) is represented as a directed acyclic graph (DAG) of 2-input AND nodes, with possibly inverted edges. Numerous algorithms based on AIGs have been developed and implemented in an academic state-of-the-art logic synthesis tool ABC [19]. Most probably, AIGs are (or soon will be) incorporated in commercial tools as well [20]. The authors of these algorithms and tools rightfully claim, that they are scalable [21].

For many years it seemed that AIGs are the ultimate solution to a network representation. However, new and more complex representations emerged recently. Bi-conditional BDDs (BBDDs) based on equivalence functions have been proposed as an alternative to BDDs. Here, simple multiplexers having one control input were replaced by equivalence functions with two controlling inputs [22]. Majority-Inverter-Graphs (MIGs) [23] have been proposed as an alternative to AIGs. Here the two-input AND nodes were replaced by three-input majority functions. Both are generalizations of the original structures, thus they are no less scalable. The primary motivation behind introducing these representations was in "emerging technologies" [24].

It has been observed that the above mentioned structures and algorithms based on them do not efficiently cope with XOR gates [25]. Particularly, most algorithms are based on AND and OR gates and heavily rely on their properties. However, XOR gates are somewhat special. They are difficult to be *identified* in the former structures (SOP, BDD, AIG) and most importantly, the algorithms do not treat them *explicitly*; typically they are identified in the technology mapping phase only. Even though there do exist algorithms performing XOR decomposition [14], [26], [27], no global network processing algorithm assumes XORs explicitly. This may yield in a lack of performance of tools, especially for XOR-intensive circuits [28], [25].

In ABC, bad synthesis performance [28] for XOR-intensive circuits can be either in algorithms inability to utilize XORs and considering AIG as a pure network of ANDs and inverters, or in XOR identification in the network.

Recently, ABC9 package with a new AIG manager called GIA has been added to ABC by its authors, introducing a possibility to use XOR or MUX nodes in addition to standard AND nodes. This package is however poorly documented and most of synthesis algorithms do not use the GIA manager. These structures directly reflect the behavior of possibly new technological primitives, thus developing algorithms based on these structures may open new ways of future logic synthesis [24].

Based on this internal network representation, we introduce XOR-AND-Inverter Graphs (XAIGs), as an alternative to

standard AIGs. Here the Boolean network is represented as a DAG of two types of nodes: 2-input ANDs and 2-input XORs, with possibly inverted edges.

Note that XAIGs represent an orthogonal approach to Majority-Inverter-Graphs (MIGs) [23]. The majority function (as well as AND) is monotonic, while XOR is *not monotonic*. Therefore, XAIGs do cover *all* relevant classes of NPN equivalence [29], [25].

To address the issue of algorithms inability to work with XORs natively, we present an XAIGs-based rewriting algorithm [18]. We have implemented it as a command in the ABC9 tool [19]. The experimental results show that XAIG-based rewriting can lead to finding additional XORs in a network as well as it can help mappers to reduce the area or delay, although it does not provide ultimate solution for all circuits.

The paper is organized as follows: after the Introduction and some preliminaries in Section II, the proposed XAIG structure is described in Section III, with implementation issues presented in Section IV. The newly introduced rewriting algorithm based on the XAIG structure is presented in Section V. Section VI contains experimental results. Section VII concludes the paper.

II. Preliminaries

And-Inverter Graphs (AIGs) [16], [17], [18], are directed acyclic graphs with one or more roots, where nodes are two-input AND gates and edges represent connections between them. Edges may be inverted, meaning that the respective subgraph is negated. This can be understood as an inverter presence on the connection. AIGs are constructed from primary inputs to primary outputs, assigning to each node a unique ID in increasing order. This ensures parent nodes to have higher ID than their children. The node with lower ID is always the left child of its parent. Apart from that, upon node creation, a hash is calculated from hashes of its children. If a node with the same hash is already present in the graph, this existing node is used by the reference instead of creating a new node. This process is called structural hashing (*"strashing"*) [16] and ensures that there will be no structurally equivalent subgraphs in an AIG.

Structural hashing still does not discover functionally equivalent subgraphs with different structures. ABC provides functionally reduced AIGs, FRAIGs [30]. If this approach (colloquially called "fraiging") is used in addition to structural hashing, also functional hasing of small subgraphs is performed.

A cut of a node N is a set of nodes (called *leaves*), for which it holds that every path from primary inputs to the node N leads through at least one *leaf*. A cut is *K-feasible* if the number of leaves does not exceed K.

III. The XAIG Structure

XAIG is a directed acyclic graph, where nodes are two-input ANDs or XORs, while edges can be inverted. As seen in Figure 1, XOR is described by at least 3 AND nodes in AIG, which can make it more difficult for algorithms to utilize it. In XAIGs, XOR is represented as a single node.

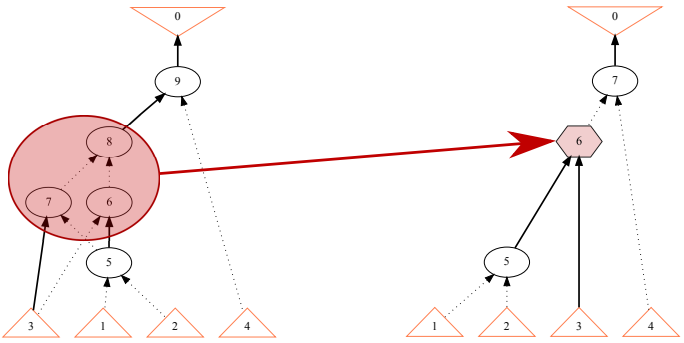

Fig. 1: Logic function $Y = \neg(\neg(x_1 \vee x_2) \oplus x3) \wedge \neg x_4$ in AIG and XAIG. Circle nodes represent AND nodes, hexagon a XOR node. Dotted edges are inverted.

A. XAIG Properties

XAIGs are a generalized form of AIGs; every AIG can be considered as an XAIG with no XOR nodes. Therefore, XAIG (just like AIG) can implement any logic function.

XAIG is structurally hashed in the same way as AIG, with different hashing function for XOR nodes. The fraiging technique is applicable also for XAIGs.

IV. Implementation of XAIGs in ABC

The ABC9 package features a new manager for AIG manipulation, called GIA manager. It has a possibility to use nodes of type XOR and MUX in addition to standard AND nodes. Therefore, we used this package and GIA as a manager for XAIG. A network can be converted between managers by the command *&get* to convert it from the original AIG manager to GIA and *&put* to convert it back.

A. The XIAG File Format

For the need of storing XAIGs in a file with unchanged structure, we defined the XAIGER file format based on the AIGER format [31] and implemented its support to ABC9 network reading and writing commands, *&r* and *&w*. The header of XIAG is described as follows:

xaig M I L O A X, where

- $M = I + L + A + X$,
- I stands for the number of inputs,
- L stands for the number of latches,
- A stands for the number of ands,
- and X stands for the number of XORs.

The nodes themselves are defined in the same way as in the original AIGER, XORs are distinguished from ANDs by having the left child node ID higher than the right one, which is forbidden in the original AIG. As an example, XOR in the XAIGER format can be described in the following way:

```
xaig 3 2 0 1 0 1
2
4
6
6 2 4
```

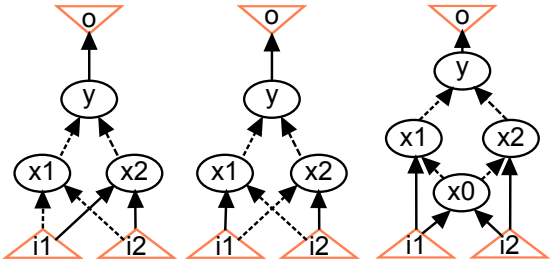

Fig. 2: XOR structures in AIG. Two leftmost structures, composed from 3 ANDs can be identified by ABC9 structural hashing. The rightmost graph represents the simplest XOR structure not identified by structural hashing in ABC9.

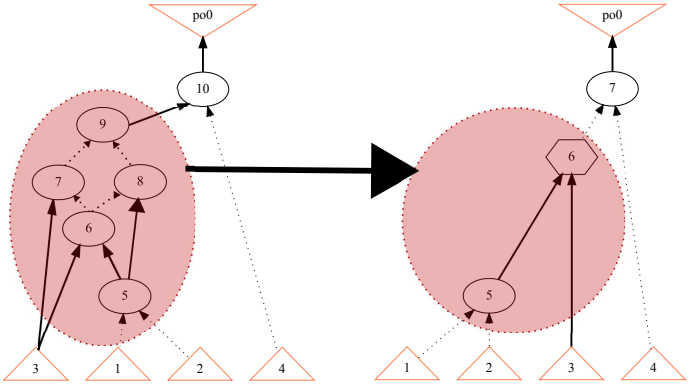

Fig. 3: XAIG based rewriting example.

This format seems to be no more necessary, as we found out that if XAIG is converted to the original AIG manager with the command *&put* and then stored to a BLIF file, it can be later reconstructed by loading and converting back to GIA by XOR-supporting structural hashing (ABC command *&st*). However, one can never be sure that the structural hashing will reconstruct the original XAIG structure exactly.

B. Recognizing XOR Gates

XOR identification is performed in the ABC9 package by structural hashing (command *&st -m -L 1*). While the parameter *-m* enables conversion to "large" gates (XOR, MUX), the parameter -L sets the reference limit for enabling generation of MUX nodes. We have implemented a new feature in ABC9, where setting this limit to 1 disables MUX creation completely.

This procedure identifies XOR gates represented by 3 ANDs as described in Figure 2. If XOR is dissolved to a flatter structure (i.e. XOR of another functions expressed as a sum of products), the *&st* command is too weak to identify it and synthesis will be unable to use it, unless it finds it by a different way, for example by a functional checking of a subtree, which we implemented in our *&rewrite* command.

V. XAIG-Based Rewriting Algorithm

To demonstrate whether synthesis needs to be capable of a native work with XOR gates, we introduce an XAIG rewriting algorithm based on AIG rewriting technique presented in [18].

Listing 1: Rewrite over XAIG network

```
Rewriting (network XAIG, node startNode, hash table
    PrecomputedStructures, bool UseZeroCost)
{
  for each node N topologically ordered after
      startNode in the network XAIG
  {
    bestXAIG = NULL; BestGain = -1;
    for each 4-input cut C of node N computed using
        cut enumeration {
      F = Boolean function of N in terms of the
          leaves of C
      // get best cut implementation
      bestCircuit =
          HashTableLookup(PrecomputedStructures, F);
      // get XAIG with cut replaced by best circuit
      rwrXAIG = ReplaceCutByBestCircuit(XAIG,
          bestCircuit, cut);
      Gain = NodesCount(XAIG) - NodesCount(rwrXAIG);
      // keep track of best possible rewrite for
          current node
      if (Gain > 0 || (Gain = 0 && UseZeroCost)){
        if (bestXAIG = NULL || BestGain < Gain){
          bestXAIG = rwrXAIG; BestGain = Gain;
        }
      }
    }
    if (bestXAIG != NULL){
        return Rewriting(bestXAIG, N+1,
            PrecomputedStructures, UseZeroCost);
    }
    else {
        continue;
    }
    return XAIG;
  }
}
```

Rewriting is a technique of replacing AIG subgraphs with k leaves (k-feasible cuts [32]) by smaller, functionally equivalent precomputed structures. This algorithm can reduce functionally equivalent subgraphs, unlike structural hashing can. An example of one subgraph replacement can be seen in Figure 3.

As described in Listing 1, the newly introduced algorithm *&rewrite* goes through XAIG nodes in topological order from defined starting node. For each node, cuts are enumerated using the algorithm presented in [32], described in Listing 2. For each node cut, a truth table of the function of its leaves is calculated by simulation. This truth table is then converted to a canonical form described by a 16-bit integer value, which is stored in a precomputed table for each possible function, so are the permutations of inputs and negations of inputs and outputs needed for this conversion. Conversion to the canoncial form is done by the same conversion table already available for the original rewriting. For every truth table in canonical form, there is a precomputed "best structure". The number of nodes saved in case of replacement of a cut by this structure is calculated. Replacement with the best number of nodes saved is performed for each node and rewriting continues with the next node in topological order. Apart from using XOR nodes in addition to AND nodes and different "best structures",

we expect no significant differences from the original *rewrite* algorithm behaviour.

Note that for 4-feasible cuts, there are 2^{16} possible functions, but every possible cut can be converted by permutation of inputs and negation of inputs and output to one of 222 NPN equivalent classes [18]. Therefore, using 4-feasible cuts for rewriting is a good compromise between the efficiency of the algorithm and its memory demands.

Listing 2: Cut enumeration algorithm

```
void NetworkKFeasibleCuts (Graph g, int k){
  for each primary output node n of g {
    NodeKFeasibleCuts(n, k);
  }
}

cutset NodeKFeasibleCuts (Node n, int k){
  if (n is primary input) return {{n}};
  if (n is visited) return NodeReadCutSet(n);
  mark n as visited;
  cutset Set1 =
      NodeKFeasibleCuts(NodeReadChild1(n), k);
  cutset Set2 =
      NodeKFeasibleCuts(NodeReadChild2(n), k);
  cutset Result = MergeCutSets(Set1, Set2, k) U {n};
  NodeWriteCutSet(n, Result);
  return Result;
}

cutset MergeCutSets (cutset Set1, cutset Set2, int
    k){
  cutset Result = {};
  for each cut Cut1 in Set1{
    for each cut Cut2 in Set2{
      if (|Cut1 U Cut2| <= k){
        Result = Result U {Cut1 U Cut2};
      }
    }
  }
  result Result;
}
```

A. Best logic structures generation

All NPN equivalent classes of cuts mentioned above have already been enumerated in ABC for the original *rewrite* algorithm. The XAIG based rewriting algorithm however needs different "best structures" than the original AIG rewriting, i.e., with XOR nodes allowed. For every NPN class, we calculated this structure by the following sequence of ABC commands: *read_truth; st; dch; if; mfs; st; dch; if; mfs; st; st; dch; if; mfs; st; dch; map; write_blif* to optimize the initial representation described by a truth table and map it using a custom library providing ANDs, XORs, and inverters. ANDs and XORs have the same cost during the mapping to not prefer any gate. The final sequence of commands just converts the mapped structure generated for each NPN class to the XAIGER format to be used with the *&rewrite* command: *read_blif; &get -m; &st -m; &w*.

VI. Experimental Results

As a comparison of AIG based synthesis performance with XAIG based synthesis, we have run both algorithms over a set of 490 benchmark circuits obtained as a mix of different benchmarks: LGSynth'91 [33], IWLS'93 [34], ISCAS'85 [35], ISCAS'89 [36], and IWLS 2005 [37] - available from [38].

A. Comparison of Rewriting Algorithms

Here we have made a comparison of the original AIG rewriting algorithm with the XAIG rewriting algorithm. Particularly, ABC commands *rewrite* and *&rewrite*) were compared. The counts of resulting nodes (AND, XOR) were measured.

The results are shown in Table I. After the circuit name, the AND and XOR nodes counts obtained from the original circuit by XOR-supporting structural hashing (the *&st -m -L 1;* command) are given. Then, results obtained by the proposed *&rewrite* command are shown. For comparison with the original *rewrite*, the numbers of AND nodes produced by it are shown next. Finally, the total cost computation is provided. Here a XOR node cost in XAIG has been counted as 3 AND nodes, because in the worst case this XOR node would be replaced by 3 AND nodes by conversion to AIG.

We can see that in most cases the total cost of nodes is higher for XAIG based rewriting than for AIG based rewriting. The reason for this seems to be that XAIG does not distinguish the cost of XOR and AND during cut replacement evaluation. Despite of this, for some circuits the total cost of XAIG based rewriting was significantly lower. In addition to this, for some circuits new XORs were discovered (see, e.g., the ripple-carry adders, where their XOR-based structure was found), while for another circuits some disappeared.

However, the declared XOR cost compared to ANDs is the worst case possible. Therefore, the improvement is more significant after a mapping depending on XOR cost in the target technology, or if some structural sharing between XORs and ANDs was found during the mapping. Also note that higher total cost of all nodes can help the mapper to not get stuck in a local optimum.

B. The Overall Synthesis Process

To demonstrate the influence of XAIG based rewriting in the overall synthesis process, we compared both rewriting algorithms by the number of LUTs after FPGA (4-LUT) mapping. The commands used for both AIG based process were *rewrite; balance; if; mfs*, which was the same for the XAIG based process except of using *&rewrite* instead of *rewrite*. Results in Table II show that for some circuits, XAIG based process produced smaller area, while other circuits work better with AIG based process. The delay after mapping is the same for both algorithms in most cases, but for some circuits there is a significant difference between the two algorithms. This is an impulse to future research on identification of the cause of this phenomenon and characterization of circuits working better with XAIG based rewriting than with AIG based rewriting.

VII. Conclusion

The answer to the question stated in the title is: "yes". There are cases, where XOR based synthesis really does help.

TABLE I: Comparison of XAIG rewriting (*&rewrite*) to AIG rewriting (*rewrite*) regarding the total cost of nodes. Only 10 circuits with the best cost ratio and 10 benchmarks with worst ratio are shown due to limited paper size. The average is calculated from all of the 490 benchmark circuits.

name	original stats		&rewrite			rewrite	rewrite; XAIG strashed			&rewrite. vs rewrite ratio
	and	xor	and	xor	cost	and	and	xor	cost	
c1355	510	0	130	59	307	446	278	56	446	0.69
i8	3090	0	1579	0	1579	1781	1781	0	1781	0.89
Altera_oc_ssram	388	0	350	0	350	388	388	0	388	0.90
majority	17	0	12	0	12	13	13	0	13	0.92
Altera_fip_cordic_cla	2130	177	1041	202	1647	1773	1178	209	1654	0.93
cm152a	31	0	27	0	27	29	29	0	29	0.93
mc	18	0	15	0	15	16	16	0	16	0.94
cm85a	32	4	31	1	34	36	28	4	40	0.94
dc2	126	2	88	4	100	105	97	3	106	0.95
cmb	54	0	45	0	45	47	47	0	47	0.96
i5	335	0	285	0	285	223	223	0	223	1.28
13-adder	91	13	80	19	137	107	55	25	130	1.28
12-adder	84	12	76	17	127	99	51	23	120	1.28
15-adder	105	15	92	22	158	123	63	29	150	1.28
14-adder	98	14	88	20	148	115	59	27	140	1.29
Altera_ts_mike_fsm	71	3	49	3	58	45	36	4	48	1.29
16-adder	112	16	100	23	169	131	67	31	160	1.29
newapla1	39	0	36	0	36	27	27	0	27	1.33
x1	1468	0	1206	0	1206	899	899	0	899	1.34
too_large	7771	0	6331	0	6331	4486	4486	0	4486	1.41
AVERAGE										**1.06**

TABLE II: Comparison of XAIG rewriting (*&rewrite*) to AIG rewriting (*rewrite*), after mapping to LUTs. Only 10 circuits with the best LUT ratio and 10 benchmarks with worst ratio are shown due to limited paper size. The average is calculated from all of the 490 benchmark circuits.

name	LUTs		Levels		&rewrite vs. rewrite ratio	
	&rw	rewrite	&rewrite	rewrite	LUTs	Levels
cm85a	10	16	3	3	0.63	1.00
majority	2	3	2	2	0.67	1.00
mm30a	241	314	33	51	0.77	0.65
02-adder	4	5	2	2	0.80	1.00
03-adder	6	7	3	3	0.86	1.00
i2	63	72	5	5	0.88	1.00
count	37	42	6	6	0.88	1.00
root	74	83	5	5	0.89	1.00
s838	111	124	12	12	0.90	1.00
Altera_ts_mike_fsm	18	20	2	2	0.90	1.00
ldd	38	34	4	4	1.12	1.00
frg1	177	156	6	6	1.13	1.00
rd53	33	29	4	4	1.14	1.00
too_large	1686	1429	8	9	1.18	0.89
bigkey	1642	1348	3	3	1.22	1.00
Altera_xbar_16x16	336	272	4	3	1.24	1.33
cm163a	14	11	3	3	1.27	1.00
mm4a	86	66	10	10	1.30	1.00
Altera_mux8_64bit	1283	963	3	3	1.33	1.00
Altera_mux8_128bit	2563	1923	3	3	1.33	1.00
AVERAGE					**1.00**	**1.01**

To prove this, we implemented an XAIG based rewriting algorithm in the framework of logic synthesis and optimization tool ABC. Experimental comparisons show that XAIG based rewriting is not the ultimate solution to the synthesis process, but it can still be beneficial in sense of area reduction, and in some cases even in delay optimization.

For some circuits, improvement can be seen on two levels: first, directly after the rewriting algorithm, the total cost of the network was smaller when XAIG based rewriting was used. Secondly, for some circuits, mapping resulted in lower area after using the XAIG based algorithm, even if the total cost of network was greater with XAIG based rewriting than with the original AIG based one.

These results point out an area of possible future research on classification of circuits, where original AIG based rewriting works better than the XAIG based rewriting, to bring a unified process advantageous for both classes.

ACKNOWLEDGMENT

This research has been partially supported by the grant GA16-05179S of the Czech Grant Agency, Fault-Tolerant and Attack-Resistant Architectures Based on Programmable

Devices: Research of Interplay and Common Features (2016-2018).

Access to computing and storage facilities owned by parties and projects contributing to the National Grid Infrastructure MetaCentrum, provided under the programme "Projects of Large Research, Development, and Innovations Infrastructures" (CESNET LM2015042), is greatly appreciated.

REFERENCES

[1] E. McCluskey, "Minimization of Boolean functions," *The Bell System Technical Journal*, vol. 35, no. 6, pp. 1417–1444, Nov 1956.

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

[3] R. Rudell and A. Sangiovanni-Vincentelli, "Multiple-valued minimization for PLA optimization," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 6, no. 5, pp. 727–750, September 1987.

[4] O. Coudert, "Doing two-level logic minimization 100 times faster," in *Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, ser. SODA'95. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 1995, pp. 112–121.

[5] G. D. Hachtel and F. Somenzi, *Logic Synthesis and Verification Algorithms*. Norwell, MA, USA: Kluwer Academic Publishers, 1996.

[6] S. Hassoun and T. Sasao, *Logic Synthesis and Verification*. Kluwer Academic Publishers, 2002.

[7] E. Sentovich, K. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. Stephan, R. K. Brayton, and A. L. Sangiovanni-Vincentelli, "SIS: a system for sequential circuit synthesis," EECS Department, University of California, Berkeley, Tech. Rep. UCB/ERL M92/41, 1992.

[8] M. Gao, J.-H. Jiang, Y. Jiang, Y. Li, A. Mishchenko, S. Sinha, T. Villa, and R. Brayton, "Optimization of multi-valued multi-level networks," in *32nd IEEE International Symposium on Multiple-Valued Logic (ISMVL'02)*, 2002, pp. 168–177.

[9] S. Muroga, Y. Kambayashi, C. Lai, Hung, and N. Culliney, Jay, "The transduction method-design of logic networks based on permissible functions," vol. 38, no. 10, pp. 1404–1424, 10 1989, iEEE Transactions on Computes.

[10] S. B. Akers, "Binary decision diagrams," *IEEE Transactions on Computers*, vol. 27, no. 6, pp. 509–516, Jun. 1978.

[11] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Transactions on Computers*, vol. 35, no. 8, pp. 677–691, Aug. 1986.

[12] K. Karplus, "Using if-then-else DAGs for multi-level logic minimization," in *Proc. of Advance Research in VLSI, C. Seitz Ed.* MIT Press, 1989, pp. 101–118.

[13] Y.-T. Lai, M. Pedram, and S. B. K. Vrudhula, "BDD based decomposition of logic functions with application to FPGA synthesis," in *Proceedings of the 30th International Design Automation (DAC'93)*. New York, NY, USA: ACM, 1993, pp. 642–647.

[14] C. Yang and M. Ciesielski, "BDS: a BDD-based logic optimization system," vol. 21, no. 7, pp. 866–876, 08 2002, iEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems.

[15] N. Vemuri, P. Kalla, and R. Tessier, "BDD-based logic synthesis for LUT-based FPGAs," *ACM Transactions on Design Automation of Electronic Systems*, vol. 7, no. 4, pp. 501–525, 12 2001.

[16] A. Kuehlmann, V. Paruthi, F. Krohm, and M. Ganai, "Robust boolean reasoning for equivalence checking and functional property verification," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 21, no. 12, pp. 1377–1394, 2001.

[17] P. Bjesse and A. Borlv, "DAG-aware circuit compression for formal verification," in *IEEE/ACM International Conference on Computer-Aided Design*, 2004, pp. 42–49.

[18] K. Brayton, Robert, A. Mishchenko, and S. Chatterjee, "DAG-aware AIG rewriting: a fresh look at combinational logic synthesis," in *43rd ACM/IEEE Design Automation Conference*. ACM, 2006, pp. 532–535.

[19] A. Mishchenko *et al.*, "ABC: A system for sequential synthesis and verification," 2012. [Online]. Available: http://www.eecs.berkeley.edu/˜alanmi/abc

[20] R. Brayton and A. Mishchenko, "ABC: an academic industrial-strength verification tool," in *Proceedings of the 22nd International Conference on Computer Aided Verification (CAV'10)*. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 24–40.

[21] K. Brayton, Robert and A. Mishchenko, "Scalable logic synthesis using a simple circuit structure," in *International Workshop on Logic and Synthesis*, 2006, pp. 15–22.

[22] L. Amaru, P.-E. Gaillardon, and G. De Micheli, "Biconditional binary decision diagrams: A novel canonical logic representation form," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 4, no. 4, pp. 487–500, Dec 2014.

[23] ——, "Boolean logic optimization in majority-inverter graphs," in *52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, June 2015, pp. 1–6.

[24] L. Amaru, P.-E. Gaillardon, S. Mitra, and G. De Micheli, "New logic synthesis as nanotechnology enabler," *Proceedings of the IEEE*, vol. 103, no. 11, pp. 2168–2195, Nov 2015.

[25] J. Schmidt and P. Fiser, "The case for a balanced decomposition process," in *12th Euromicro Conference on Digital System Design, Architectures, Methods and Tools*, Aug 2009, pp. 601–604.

[26] A. Mishchenko and M. Perkowski, "Fast heuristic minimization of exclusive-sums-of-products," in *International Workshop on Reed-Muller expansions in circuit design*, 2001, pp. 242–249.

[27] A. Mishchenko, M. Perkowski, and B. Steinbach, "An algorithm for bi-decomposition of logic functions," in *38th ACM/IEEE Design Automation Conference*, 2001, pp. 103–108.

[28] P. Fiser and J. Schmidt, "Small but nasty logic synthesis examples," in *8th. Int. Workshop on Boolean Problems (IWSBP)*, 2008, pp. 183–189.

[29] Z. Huang, L. Wang, Y. Nasikovskiy, and A. Mishchenko, "Fast boolean matching based on NPN classification," in *International Conference on Field-Programmable Technology (FPT)*, Dec 2013, pp. 310–313.

[30] A. Mishchenko, S. Chatterjee, R. Jiang, and K. Brayton, Robert, "Fraigs: A unifying representation for logic synthesis and verification," Berkeley University, Tech. Rep.

[31] A. Biere, "AIGER," http://fmv.jku.at/aiger/, 2007.

[32] A. Mishchenko, S. Chatterjee, K. Brayton, Robert, X. Wang, and T. Kam, "Technology mapping with boolean matching, supergates and choices," ERL Technical Report, EECS Dept., UC Berkeley, Tech. Rep., 03 2005.

[33] S. Yang, "Logic synthesis and optimization benchmarks user guide: Version 3.0," MCNC Technical Report, Tech. Rep., Jan. 1991.

[34] K. McElvain, "IWLS'93 Benchmark Set: Version 4.0," Tech. Rep., May 1993.

[35] F. Brglez and H. Fujiwara, "A Neutral Netlist of 10 Combinational Benchmark Circuits and a Target Translator in Fortran," in *IEEE International Symposium Circuits and Systems (ISCAS'85)*. IEEE Press, Piscataway, N.J., 1985, pp. 677–692.

[36] F. Brglez, D. Bryan, and K. Kozminski, "Combinational profiles of sequential benchmark circuits," in *IEEE International Symposium on Circuits and Systems (ISCAS'89)*, May 1989, pp. 1929–1934 vol.3.

[37] C. Albrecht, "IWLS 2005 benchmarks," Tech. Rep., Jun. 2005.

[38] S. J. Fiser P., "A comprehensive set of logic synthesis and optimization examples," pp. 151–158, 2016. [Online]. Available: http://ddd.fit.cvut.cz/prj/Benchmarks/