Journal of Circuits, Systems, and Computers
Vol. 28, Suppl. 1 (2019) 1940010 (29 pages)

DOI: 10.1142/S0218126619400103

SAT-Based Generation of Optimum Circuits with Polymorphic Behavior Support*

Petr Fišer†, Ivo Háleček‡ and Jan Schmidt§

Faculty of Information Technology,
Czech Technical University in Prague,
160 00 Praha 6, Czech Republic
†*fiserp@fit.cvut.cz*
‡*halecivo@fit.cvut.cz*
§*schmidt@fit.cvut.cz*

Václav Šimek

Faculty of Information Technology,
Brno University of Technology,
612 66 Brno, Czech Republic
simekv@fit.vutbr.cz

Received 14 October 2018
Accepted 20 April 2019
Published 14 June 2019

This paper presents a method for generating optimum multi-level implementations of Boolean functions based on Satisfiability (SAT) and Pseudo-Boolean Optimization (PBO) problems solving. The method is able to generate one or enumerate all optimum implementations, while the allowed target gate types and gates costs can be arbitrarily specified. Polymorphic circuits represent a newly emerging computation paradigm, where one hardware structure is capable of performing two or more different intended functions, depending on instantaneous conditions in the target operating environment. In this paper we propose the first method ever, generating provably size-optimal polymorphic circuits. Scalability and feasibility of the method are documented by providing experimental results for all NPN-equivalence classes of four-input functions implemented in AND–Inverter and AND–XOR–Inverter logics without polymorphic behavior support being used and for all pairs of NPN–equivalence classes of three-input functions for polymorphic circuits. Finally, several smaller benchmark circuits were synthesized optimally, both in standard and polymorphic logics.

Keywords: Boolean functions; logic synthesis; SAT; PBO; optimum implementation; exact synthesis; polymorphic circuits.

*This paper was recommended by Regional Editor Zoran Stamenkovic.
§Corresponding author.

1. Introduction

The need for obtaining optimum gate-level implementations of Boolean functions is encountered in many applications, mostly in logic synthesis and optimization. The rewriting algorithm,[1] where subcircuits of a network are repetitively replaced by their (preferably) *optimum* implementations, is the most striking example. The optimum can be understood in terms of area, but the delay could be of concern as well.

In the original rewriting algorithm,[1] the replacement circuits were produced by a branch-and-bound algorithm, which is a rather time-consuming process, and for this reason the optimum implementations of all functions (or their NPN classes, respectively[2–4]) of a given number of inputs are precomputed and stored in memory. Moreover, this approach is not extensible; once new technological primitives are to be considered or different primitives' costs are to be assumed, all the implementations must be recomputed.

Having a single optimum implementation of each function is not sufficient. Particularly, even though all optimum implementations (in size and delay) are equal as stand-alone solutions, their incorporation in a bigger network may make a difference, mostly because of logic sharing possibilities.[1,5]

For the above reasons, a more universal, flexible, and scalable way of computing optimum implementations is required. In this paper, we propose a method able to use any technological primitives (logic gates) for the implementation, assign arbitrary costs to gates and thereby compute the optimality, generate one or more (or all) solutions, and run on demand, because of its relatively low runtime. The proposed algorithm also supports the design of *polymorphic circuits*, where one hardware structure generates two different functions, based on external stimuli, see Sec. 1.1.

In its basic principle, the method is based on generating a Satisfiability (SAT) problem instance in a conjunctive normal form (CNF) for a given function, whose solution is the optimum function implementation. Next, Pseudo-Boolean Optimization (PBO)[6] is incorporated to accommodate different gate costs. Enumeration SAT problem solving can be used to obtain more or even all solutions.

The decision version of the Optimum Circuit Problem is Σ_2^P-complete,[7,8] when the instance size is measured by the number of variables. Since the suggested problem exhibits such an immensely complex nature, it is possible to design provably optimum implementations only for functions with a severely limited number of inputs, typically up to 10. However, this cannot be overcome now by any means (unless a significant breakthrough in circuit complexity theory happens).

The contributions of this paper can be summarized as follows:

(1) SAT- and PBO-based algorithm for optimum circuits synthesis. Therefore, complex formal methods like Satisfiability Modulo Theories (SMT)[9] with slow solvers are avoided.

(2) Support for any type of nodes (gates). That is, the optimum circuit can be constructed of any gates, including those implementing nonsymmetric functions. Nodes of individual gate type may be assigned a specific cost.
(3) Multi-output functions support.
(4) Polymorphic behavior support. It is implemented by polymorphic edges. Such an approach we have adopted is, indeed, a generalization of possible polymorphic gates.
(5) The ability of the algorithm to enumerate all solutions.
(6) Support for depth optimization. Since the total network cost is the primary optimality criterion, the network depth (number of levels) can be assigned as the secondary.

This paper is an extended version of Refs. 10 and 11. In this paper, the SAT instances generation is described in a more detailed way, as well as the overall algorithm with all its modifications (influence of gates cost, enumeration of all solutions, and depth minimization). Also, more comprehensive experimental results are presented.

The paper is structured as follows: after the introduction, the related works on optimum circuits design and polymorphic circuits design are presented in Sec. 2. Section 3 gives the preliminaries necessary for understanding the rest of the paper. The proposed method is described in detail in Sec. 4. The algorithm is experimentally evaluated in Sec. 5. Section 6 concludes the paper.

1.1. *Polymorphic circuits*

The still ongoing trend of relentlessly scaling diverse circuit features in close compliance with Moore's law, while the ubiquitous CMOS technology is approaching its technology limits at the same time, is raising the necessity to introduce rather unconventional technological solutions accompanied by a range of suitable computational paradigms.[12] Thus, it becomes obvious that so-called emerging technologies and their properties will play a substantial role in pursuing a new generation of devices. Polymorphic circuits can be recognized as one of such examples.[13] Here, using a single hardware structure, two or more intended functions can be implemented. Selection mechanism of the active function at a given moment in time involves the natural occurrence of external conditions, like temperature, supply voltage, light, or even an additional control signal, which have a direct impact on the electrical properties of particular hardware circuitry.

The motivation behind the activities ultimately resulting in the creation of polymorphic circuits (or similar ones concerning the general principle of operation) has been originally given by the need to address the following aspects properly: (1) autonomous temperature compensation of a system's properties once it is deployed in harsh environments[14] without the possibility of performing regular maintenance, (2) "graceful degradation" of a system;[15] that ensures, for example, its safe shutdown or triggers its automatic transition into a low-power emergency

operating mode in case of low or depleted batteries. Another application field can be identified in connection with security devices — a hidden function (watermark) can be implemented using polymorphic electronics.[15]

The first attempt to design such circuits has been made by Stoica *et al.* from NASA's Jet Propulsion Laboratory, for adaptively changing the function of a device based on the environment temperature.[13–17] MOS transistor structures were proposed to accomplish this job and polymorphic gates were developed and manufactured, performing distinctive logic functions (e.g., NAND/NOR,[16] AND/OR[17]) once exposed to variable temperature ranges.

Due to fundamental constraints projected in the capabilities of existing conventional methods in terms of enabling the design of such structures,[15] these polymorphic gates were mostly generated by suitable evolutionary techniques. In particular, the application of genetic programming[14,16,17] took place. At present, it is possible to design virtually any type of polymorphic gate by using these techniques.[18]

Theoretical backgrounds and most importantly the physical availability of polymorphic gates open a path towards the opportunity to actually design even larger circuits comprising more than just a few logic gates. From the practical point of view, it makes no sense indeed to employ the polymorphic logic gates with the sensitivity to the external conditions within a given circuit solely; a typical circuit with multi-functional or polymorphic behavior is always built around the combination of conventional, mono-functional logic gates, while their polymorphic counterparts are used only for a portion of the circuit structure. Numerous attempts to design polymorphic circuits of an arbitrary size have been already reported.[18–21] However, all these approaches are considerably suffering from non-optimality of the resulting circuit structure; no obvious proofs to validate the optimality or at least a lower bound imposed on the size complexity have been demonstrated.

2. Related Works

2.1. *Optimum circuits generation*

The problem of obtaining optimum multi-level representations of Boolean functions has been tackled since the 1960s. The optimal two-level minimization principles were extended to a multi-level domain in Ref. 22. Synthesis methods based on functional decomposition were published in Refs. 23–25. Later on, different branch-and-bound techniques were presented in Refs. 4, 26, 27, 28, and 29. All these techniques were extremely time-demanding and sometimes also memory-demanding.

Almost in parallel to this explicit track, formal (implicit) techniques were used for the purpose of optimum circuit implementations generation: approaches based on Integer Linear Programming (ILP) were proposed in Refs. 30–32. Optimum solutions based on NOR gates for functions of up to four variables were computed there.

A SAT-based approach was introduced in Ref. 33. However, it was targeted strictly to MOD-functions as nodes. This idea was later extended in Ref. 11 to support AND and XOR gates,[34,5] and in Ref. 35 to support gates of any type.

One of the recent works[36] presents optimum circuit generation for Majority-Inverter Graphs based on SMT.[9] This approach is claimed to be scalable enough to be repeatedly run in the runtime of a rewriting process.

As it can be seen from above, numerous strategies have been used to produce optimum circuit implementations. However, none of them was general enough to support any type of gates, incorporate gate costs, and was capable of enumerating all solutions. In this paper, we present a remedy to this situation.

In our approach, instead of using time-consuming ILP- or SMT-solvers, we stay with a standard (and simple) SAT-based approach, for which very efficient solvers exist.[37] Apart from SAT, a PBO-solver[38] is used for the purposes of area optimization in the presence of different gate costs. Even though the PBO solving is more time-consuming, the number of solver runs is minimized, typically to one run only.

2.2. *Polymorphic circuits design*

Based on the availability of suitable polymorphic gates, the design of more complex (larger) polymorphic circuits has been tackled recently. One of the approaches is quite straightforward — a polymorphic circuit switching between two functions can be obtained easily by means of implementing these two functions separately, whereas the actual output is selected by a single polymorphic multiplexer.[18] Another approach, based on BDDs,[39] was also proposed in Refs. 18 and 19. The method is known as PolyBDD. Its key aspect builds upon the exploitation of Multi-terminal BDD (MTBDD),[40] which is an extension of the well-established binary decision diagram concept where the terminal nodes of the diagram contain integer values. The PolyBDD method uses these values as a way of expressing a possible relation between input variables and the relevant output.

When a typical design objective calls for implementation of a pair of two intended functions denoted as, e.g., F_1 and F_2 (each of them is actively executed in one of the permissible yet mutually distinct operating modes), a corresponding MTBDD tree is created. As a next step, the MTBDD representation is converted into a target circuit structure, where its nodes assume the role of multiplexers and terminals are replaced by proper polymorphic subcircuits according to the numbers in their respective leaves. These values placed in the terminal nodes of the MTBDD tree will assume integer values from the interval 0–15 (see Fig. 1 for details) in case of functions F_1 and F_2. Detailed explanation of internal principles of the PolyBDD method can be found in Refs. 18 and 19.

Nevertheless, the principal drawback of both situations may be recognized as a relatively sparse utilization of polymorphic gates, when these have been moved just to the peripheral edge of a target circuit (practically used only in the role of

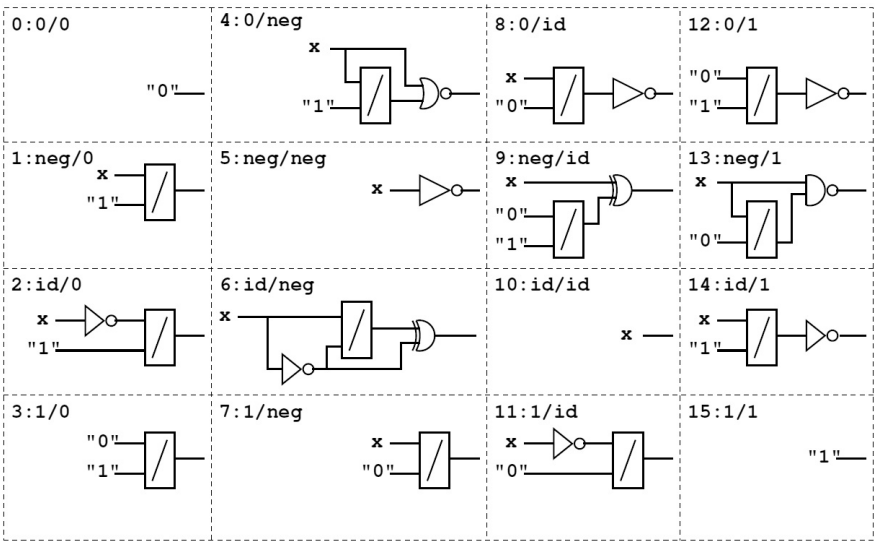

Fig. 1. Conversion table for the transformation procedure of PolyBDD into a polymorphic circuit.[19]

input/output switches). This approach eventually leads to very inefficient results, since there is no shared portion of logic between the two operating modes of a given polymorphic circuit.

The first attempt to use the sharing of logic resources between the two specified functions has been addressed in Ref. 21. Based on the initial, two-level description of these functions, shared co-kernels[41] are subsequently identified, thus making it possible to move polymorphic gates "deeper" into the circuit structure. However, it is still possible to observe the inclination to place the polymorphic gates near the circuit outputs.

The actual optimality of results obtained by the methods mentioned above is rather questionable at least. Hence, obtaining lower bounds on size complexity of polymorphic circuits, i.e., designing optimum implementations of these circuits, is a vital task now.

Even though the design of optimum circuit implementation is a mature and relatively well-mastered process in case of standard logic (see Sec. 2.1), no such approach has been proposed for polymorphic circuits until now. The remedy to this is presented in this paper, as one of its main contributions.

3. Preliminaries

3.1. *Circuit representation*

Multi-output combinational circuits will be assumed throughout the paper. A combinational logic circuit can be represented as a directed acyclic graph (DAG),

with nodes corresponding to gates (logic functions they implement) and edges representing the connections between them. The DAG has one or more roots corresponding to the circuit's primary outputs (POs) and the DAG leaves correspond to its primary inputs (PIs).

Since one of the fundamental objectives of the proposed method is to make the optimum circuit generation procedure general enough to be directly applicable to any technology, the set of node functions will not be restricted by any means. However, only two-input nodes will be assumed in this paper, for the sake of simplicity. Therefore, each node may implement *any* two-input function (out of 10 possible). Despite this limitation imposed, the method can easily be extended to support nodes with any number of inputs, without the need of introducing any additional principal modifications.

Similarly to Reduced Boolean Circuits (RBCs)[42,43] or AND–Inverter Graphs (AIGs),[1,44] the edges may be negated, to indicate the presence of an inverter at the edge.

An example DAG of a 1-bit full-adder constructed from AND and XOR gates (XAIG[5]) is shown in Fig. 2.

3.2. *Polymorphic circuits representation*

Besides the role of an ordinary negation, graph edges may assume the polymorphic nature. This means all respective (polymorphic) edges are further negated when the external polymorphic stimulus occurs. The polymorphic stimulus (denoted as P in the following text) enables the selection of the circuit operating mode (out of the two

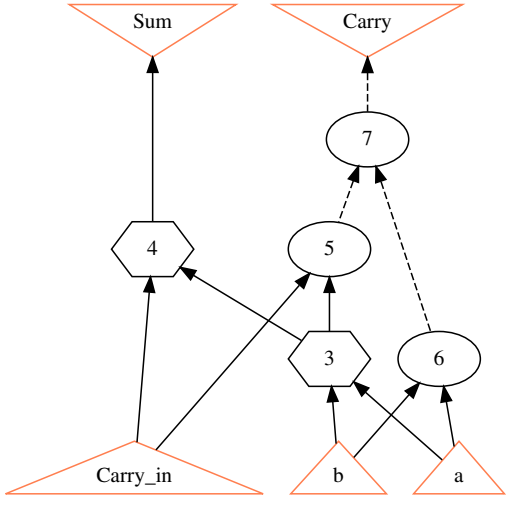

Fig. 2. A 1-bit full-adder described by an XAIG. Oval nodes represent AND gates, hexagon nodes are XORs, and the dashed edges are negated.

intended functions). As a result, there are four types of edges: normal, negated, polymorphic, and negated polymorphic ones. Assume an edge from node x to node y. Then the respective four types of edges perform these operations:

- Normal edge: $y = x$.
- Negated edge: $y = \bar{x}$.
- Polymorphic edge: $y = x \oplus P$.
- Negated polymorphic edge: $y = \bar{x} \oplus P$.

An example of such a DAG [Polymorphic XAIG (PXAIG)[45]] is shown in Fig. 3(a), for a polymorphic circuit implementing a function AND/XOR, i.e., the function

$$F = \bar{P}(a \cdot b) + P(a \oplus b)\,, \tag{1}$$

where P is the polymorphic stimulus.

In the figure, circle nodes represent AND gates, hexagon nodes represent XOR gates, dashed edges are negated, bold blue edges are polymorphic, and dashed bold blue edges are the negated polymorphic ones.

Another example is shown in Fig. 3(b), for a polymorphic 1-bit full-adder. Here, one of the adder inputs is implemented as the polymorphic stimulus. Notice that polymorphic edges actually represent implicit XOR gates (see the difference from Fig. 2). This is the first hint that efficient implementation of polymorphic behavior support may significantly reduce the amount of logic.[10]

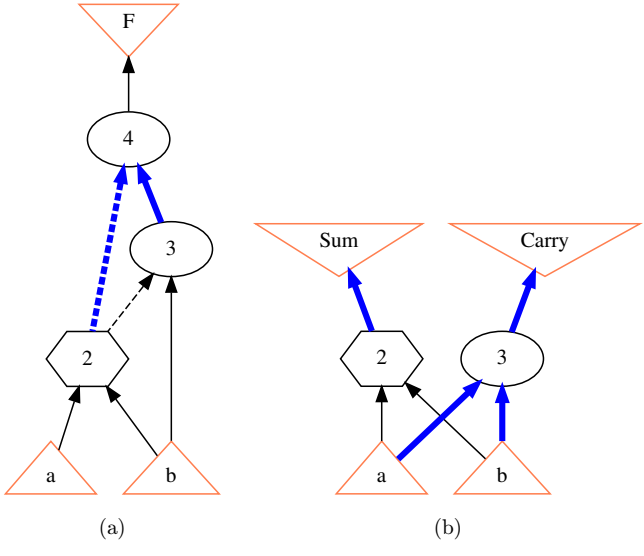

Fig. 3. Examples of polymorphic circuits represented by a DAG (PXAIG), for: (a) AND/XOR circuit and (b) polymorphic 1-bit full-adder. Polymorphic edges are in bold.

3.3. *Boolean SAT problem*

The CNF Satisfiability problem (CNF-SAT)[46] is defined as follows: given a Boolean formula in its CNF, find a satisfying assignment of its variables. A literal is a variable or its negation. A clause is a sum (disjunction) of literals. The CNF is a product (conjunction) of clauses. The formula is satisfiable when there exists an assignment of its variables so that the functional value of the formula is equal to one, i.e., each clause evaluates to one under a given assignment of variables.

The decision SAT problem just gives an answer (positive or negative) about satisfiability, whereas its constructive version returns a satisfiability witness as a result, i.e., the satisfying assignment of variables.

3.4. *PBO problem*

The PBO problem[6] can be simply understood as a special case of the ILP problem, or an extension of SAT with optimization capabilities. Instead of processing a product of clauses, a set of linear inequalities is processed. An integer-weighted sum of literals is present on the left-hand side of each inequality and integers are on the right-hand side. Generally, each PBO inequality is written in the form

$$C_0 y_0 + C_1 y_1 + \cdots + C_{n-1} y_{n-1} \geq C\,, \tag{2}$$

where y_i are Boolean variables, and C_i and C are integer constants.

The optimization criterion is defined as an integer-weighted sum of variables, i.e., similarly to the left-hand side of the inequalities. The optimization criterion is to be either minimized or maximized.

Dedicated PBO-solvers exist,[47–49] or the problem is solved by repeated application of a SAT-solver.[37,38,50]

In later sections we will need to transform a standard CNF to a PBO instance. This can be accomplished in a straightforward way:

(1) For each Boolean variable $x_0, \ldots, x_{m-1}$ of the CNF, construct an integer variable $y_0, \ldots, y_{m-1}$ of the PBO.
(2) For each CNF clause $(l_0 \vee l_1 \vee \cdots \vee l_i)$ where l_i are its individual literals (variables or their negations), construct an inequality $L_0 + L_1 + \cdots + L_j \geq 1$.
(3) For each $k \in \{0, \ldots, l\}$:

    If a literal $l_k = x_k$ (variable in its direct form), then substitute $L_i = y_k$ in the inequality.
    If a literal $l_k = \overline{x_k}$ (variable in its negated form), then substitute $L_i = (1 - y_k)$.

As an example, let us have a clause $(x_0 \vee \overline{x_1} \vee \overline{x_2})$. This clause can be transformed to a PBO inequality as follows:

$$y_0 + (1 - y_1) + (1 - y_2) \geq 1\,, \tag{3}$$

Table 1. List of all two-input functions.

Index	Gate	Function	Symmetric $a \circ b = b \circ a$	Symmetric to negation $a \circ \bar{b} = \bar{a} \circ b = \overline{a \circ b}$
0		0	Y	
1	NOR	$\overline{a+b}$	Y	
2		$\bar{a}b$		
3	NOT	$\bar{a}$		
4		$a\bar{b}$		
5		$\bar{b}$		
6	XOR	$a\bar{b} + \bar{a}b$	Y	Y
7	NAND	$\overline{ab}$	Y	
8	AND	ab	Y	
9	XNOR	$ab + \bar{a}\bar{b}$	Y	Y
A		b		
B	IMPLY	$\bar{a} + b$		
C		a		
D		$a + \bar{b}$		
E	OR	$a + b$	Y	
F		1	Y	

which is

$$y_0 - y_1 - y_2 \geq -1\,. \tag{4}$$

3.5. *List of two-input functions*

As it was stated above, the procedure described in this paper supports any two-input function as a circuit node. Just for clarity, here we present a list of all two-input functions with their symmetry properties (Table 1). These symmetries can be efficiently used in the algorithm to prune the search space, see Sec. 4.3. The functions' indexes are derived from their truth tables (see, e.g., Ref. 4).

We can also notice that there are only 10 two-input functions, from which only three (AND, XOR, and IMPLY) are of practical use, assuming negated edges are provided (i.e., all gate inputs and outputs can be possibly further negated).

4. The Proposed Method

The proposed SAT-based method of designing size-optimal (and possibly depth-optimal) circuits will be presented in this section. Any set of two-input gates can be used as a set of building blocks (DAG nodes), with their costs (area) specified. As it was stated in Sec. 3.2, the DAG edges may be negated and/or polymorphic.

First, for better understanding, we will present the algorithm in its basic, SAT-based form, to devise an optimal DAG implementation of a given Boolean function. The algorithm will be then extended to support additional features, namely the enumeration of all solutions, depth-optimal implementations generation, and customizable gate costs. The polymorphic behavior support will be assumed throughout

the whole text. In case it is not required, the parts implementing this feature can be easily "bypassed" when implementing the algorithm.

The following basic variables will be used in the following text:

- k denotes the number of implemented function PIs,
- o denotes the number of implemented function POs,
- n denotes the number of gates in the implementation.

4.1. *The basic procedure*

The Optimum Circuit Problem is solved by its reduction to a decision CNF-SAT problem.[46] Since these problems belong to different complexity classes of polynomial hierarchy (the decision Optimum Circuit Problem is Σ_2^P-complete[7,8]), the reduction cannot be polynomial. The exponential complexity increase is caused by the enumeration of all function minterms values, as a part of the produced SAT instance.

The optimization problem is reduced to its decision version by a simple trick: a decision problem "Does there exist an n-node implementation of a given k-input function?" is solved, whereas we start with $n = 1$. If the answer is negative, n is increased. This procedure is repeated until a positive answer is obtained. The solution witness is then the optimum solution for the original problem.[36]

Note that this incremental approach is not the only one possible; binary search is one option, as also proposed in Ref. 36. However, for circuits with a relatively small number of nodes, this approach is not efficient, since obtaining the upper size bound may be excessively time-consuming.

The most basic procedure is outlined by a pseudo-code shown in Fig. 4. The input to the algorithm is a truth table (a set of $o\,2^k$ binary vectors, for an o-output function) of the function to be implemented; the output is its optimal multi-level implementation structure.

4.2. *The CNF construction*

The main procedure of the algorithm, the SAT instance generation (`Generate_CNF` in Fig. 4), is described here.

```
Generate_structure (truth_table f, int k) {
   n = 1;
   do {
      CNF = Generate_CNF(f, k, n);
      Sol = SAT_Solve(CNF);
      if (Sol.unsat) n++;
   } while (Sol.unsat);
   return Sol.extract_structure;
}
```

Fig. 4. The basic size-optimal structure generation procedure.

Let us have a polymorphic circuit with k inputs and o outputs constructed of n gates. As stated in Sec. 3.2, such a circuit can be represented as a DAG with attributed (negated and/or polymorphic) edges. The following variables and constraints are introduced:

- For some special purposes (like a constant output in a multioutput function), a constant node must be present. Therefore, the node indexed as 0 will be a constant "0". The constant "1" can be obtained from it using a negated edge;
- each PI and DAG internal node has a unique index, $1, \ldots, n+k$ where DAG PIs are represented by the nodes indexed $1, \ldots, k$ and internal nodes are indexed $k+1, \ldots, n+k$;
- the DAG has o outputs, each can be connected to any node $0, \ldots, n+k$ (i.e., also directly to the constant or a PI);
- the parent node always has a higher index than both its children;
- node inputs (exactly two here) are labeled 0 (left input) and 1 (right input);
- each node can implement any gate function from a given set of functions $\mathbf{F}$. The ordinary numbers of functions are binary-encoded, i.e., v variables are needed to index the functions, $v \in \{0, \ldots, \lceil \log_2 |\mathbf{F}| \rceil - 1\}$. Since only two-input functions are considered in this paper for simplicity, $v \in \{0, \ldots, 3\}$, to be able to encode the 10 two-input functions. In practice, the number of functions can be even less, since using only three two-input functions makes sense (see Sec. 3.5).

In order to design the desired network implementing a given Boolean function, two sets of constraints must be encoded into the SAT instance: (1) the network *structure* and (2) the network *function*, i.e., propagation of values from the PIs to POs.

For easier understanding of the following text, a sketch of the designed network (DAG) with corresponding variables describing the structure (node labels) and function (edge labels) is shown in Fig. 5 for a node i with its attributed edges (a); connection of a node j to the m^{th} input of a node i (b); and connection of a node i to the primary output w with its attributed edges (c). The circle node represents the node i, rectangle nodes describe edge value modifiers (negation, polymorphic behavior), and the diamond-shaped nodes represent the network interconnection. The meanings of labels and signals in the figure will be described in the following text.

Let us note that different structure encodings can be used, e.g., as shown in Ref. 35. In principle, different encodings offer a trade-off between the instance size and its "simplicity" for SAT-solvers. In this paper, we present the encoding we think is the most understandable to readers.

4.2.1. *Network structure description*

In the scenario given above, a set of Boolean variables describing the structure of the network is defined. Note that the limits of variables' indexes already impose some structural constraints to the network, which are necessary for its validity and unambiguity.

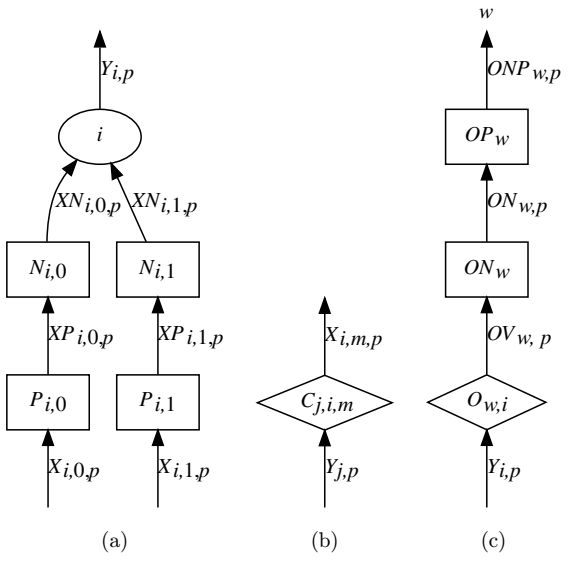

Fig. 5. Network structure with corresponding variables.

- $N_{i,m}$: The m^{th} input of node i is negated, defined for $i \in \{k+1, \ldots, n+k\}$, $m \in \{0, 1\}$.
- $P_{i,m}$: The m^{th} input of node i is polymorphic, defined for $i \in \{k+1, \ldots, n+k\}$, $m \in \{0, 1\}$.
- $C_{i,j,m}$: The output of the i^{th} node is connected to the m^{th} input of the j^{th} node, defined for $i \in \{0, \ldots, n+k-1\}$, $j \in \{\max(i, k+1), \ldots, n+k\}, m \in \{0, 1\}$.
  Note: i ranges to $n+k-1$ only, since the last node $(n+k)$ cannot be connected to any other node.
  Note: j ranges from i if i is an internal node or from $k+1$ if i is a PI or constant, to ensure that a PI would not be fed by any node.
- $O_{w,i}$: The w^{th} output is connected to the i^{th} node, $w \in \{0, \ldots, o-1\}$, $i \in \{0, \ldots, n+k\}$.
- ON_w: The w^{th} output is connected to a node by a negated edge, $w \in \{0, \ldots, o-1\}$.
- OP_w: The w^{th} output is connected to a node by a polymorphic edge, $w \in \{0, \ldots, o-1\}$.
- $F_{i,v}$: The i^{th} node function selector, $i \in \{k+1, \ldots, n+k\}$, $v \in \{0, \ldots \lceil \log_2 |\mathbf{F}| \rceil - 1\}$.

Next, constraints (SAT clauses) are defined, to describe the network structure *validity*. Note that the universal quantifiers actually represent conjunctions of SAT clauses. The final CNF-SAT instance is then formed by conjunction of all the constraints.

(1) Each node input is connected somewhere — to a node with a smaller index (including PIs and the constant):

$$\forall i \in \{k+1, \ldots, n+k\}, m \in \{0,1\} : \bigvee_{j \in \{0,\ldots,i-1\}} C_{j,i,m} \,. \tag{5}$$

(2) Each node output is connected somewhere — to a node with a higher index or to an output:

$$\forall i \in \{k+1, \ldots, n+k\} : \left(\bigvee_{\substack{j \in \{\max(i+1,k+1), \ldots, n+k\} \\ m \in \{0,1\}}} C_{i,j,m} \right) \vee \bigvee_{w \in \{0,\ldots,o-1\}} O_{i,w} \,. \tag{6}$$

(3) Each node input has only one source:

$$\forall j \in \{k+1, \ldots, n+k\}, \quad i \in \{0, \ldots, i-1\}, \\ h \in \{0; j-1\}, \quad h \neq i, \quad m \in \{0,\, 1\} : C_{i,j,m} \Rightarrow \overline{C_{h,j,m}} \,. \tag{7}$$

In simple words, if the m^{th} input of the node j is connected to the node i ($C_{i,j,m}$), it must not be connected to the node h.

This is in CNF:

$$\forall j \in \{k+1, \ldots, n+k\}, \quad i \in \{0, \ldots, i-1\}, \\ h \in \{0; j-1\}, \quad h \neq i, \quad m \in \{0,1\} : \overline{C_{i,j,m}} \vee \overline{C_{h,j,m}} \,. \tag{8}$$

Note that it is allowed to connect both node inputs to one source (the $h \neq i$ condition enables this). This feature can be used in polymorphic circuits. For example, assume a node x connected to both inputs of an AND gate, while one input edge is polymorphic. Then, the operation $x \cdot (x \oplus P) = x \cdot \bar{P}$ is performed by the AND node, which is meaningful. On the other hand, in the case of no demand for polymorphic operation, nodes connected to only one source become redundant, and from the nature of the algorithm, they will never be produced.

(4) Each primary output is connected to at least one node or primary input, or the constant:

$$\forall w \in \{0, \ldots, o-1\} : \bigvee_{i \in \{0,\ldots,n+k\}} O_{w,i} \,. \tag{9}$$

(5) Each primary output is connected to one node or primary input, or the constant at most:

$$\forall w \in \{0, \ldots, o-1\}, \quad i \in \{0, \ldots, n+k\}, \\ j \in \{i+1, \ldots, n+k\} : O_{w,i} \Rightarrow \overline{O_{wj}} \,. \tag{10}$$

In simple words, if the output w is connected to node i ($O_{w,i}$), it must not be connected to node j.

This is in CNF:

$$\forall w \in \{0,\ldots,o-1\}, \quad i \in \{0,\ldots,n+k\},$$
$$j \in \{i+1,\ldots,n+k\} : \overline{O_{w,i}} \vee \overline{O_{w,j}}\,. \tag{11}$$

4.2.2. *Network function description*

Next, the desired *function* must be enforced. This means that the network must output the correct functional value for each input combination, i.e., for all 2^k minterms for all outputs.

For this purpose, additional Boolean variables, specific for each $p \in \{0,\ldots,2^k-1\}$ minterm, are defined:

- $Y_{i,p}$ is the i^{th}, node output value, $i \in \{0,\ldots,n+k\}$.

  For $i \in \{1,\ldots,k\}$, it represents a PI.
  For $i = 0$, it is the constant "0".

- $X_{i,m,p}$ is the value of the m^{th} input of node i, defined for $i \in \{k+1,\ldots,n+k\}$, $m \in \{0,1\}$.
- $XN_{i,m,p}$ denotes the value of the m^{th} input of node i, after possible negated edge following the polymorphic edge, defined for $i \in \{k+1,\ldots,n+k\}$, $m \in \{0,1\}$.
- $XP_{i,m,p}$ is the value of the m^{th} input of node i, after possible polymorphic edge, defined for $i \in \{k+1,\ldots,n+k\}$, $m \in \{0,1\}$.
- $OV_{w,p}$ is the w^{th} output node value.
- $ON_{w,p}$ is the w^{th} output node value, after possible negation.
- $ONP_{w,p}$ is the w^{th} output node value, after possible polymorphic edge, following the negation.
- P_p denotes the polymorphic stimulus value.

Next, constraints enforcing the function are defined:

(6) Polymorphic edges:

$$\forall i \in \{k+1,\ldots,n+k\}, \quad m \in \{0,1\} :$$
$$XP_{i,m,p} = (P_{i,m} \Rightarrow P_p \oplus X_{i,m,p}) \vee (\overline{P_{im}} \Rightarrow X_{i,m,p})\,. \tag{12}$$

In simple words, in the presence of a polymorphic edge ($P_{i,m}$), negate or copy the value based on the polymorphic stimulus value (P_p). Copy the value otherwise.

This is in CNF:

$$\begin{aligned}\forall i \in \{k+1,\ldots,n+k\},\quad & m \in \{0,1\}: \\ & (\overline{P_{i,m}} \vee \overline{XP_{i,m,p}} \vee X_{i,m,p} \vee P_p) \wedge (\overline{P_{i,m}} \vee XP_{i,m,p} \vee \overline{X_{i,m,p}} \vee P_p) \\ & \wedge (\overline{P_{i,m}} \vee XP_{i,m,p} \vee X_{i,m,p} \vee \overline{P_p}) \wedge (\overline{P_{i,m}} \vee \overline{XP_{i,m,p}} \vee \overline{X_{i,m,p}} \vee \overline{P_p}) \\ & \wedge (P_{i,m} \vee X_{i,m,p} \vee \overline{XP_{i,m,p}}) \wedge (P_{i,m} \vee \overline{X_{i,m,p}} \vee XP_{i,m,p})\,. \end{aligned} \tag{13}$$

(7) Negated edges:

$$\forall i \in \{k+1,\ldots,n+k\},\quad m \in \{0,1\}: XN_{i,m,p} = XP_{i,m,p} \oplus N_{i,m}\,. \tag{14}$$

This is in CNF:

$$\begin{aligned}\forall i \in \{k+1,\ldots,n+k\},\quad & m \in \{0,1\}: \\ & (XN_{i,m,p} \vee \overline{XP_{i,m,p}} \vee N_{i,m}) \wedge (XN_{i,m,p} \vee XP_{i,m,p} \vee \overline{N_{i,m}}) \\ & \wedge (\overline{XN_{i,m,p}} \vee \overline{XP_{i,m,p}} \vee \overline{N_{i,m}}) \wedge (\overline{XN_{i,m,p}} \vee XP_{i,m,p} \vee N_{i,m})\,. \end{aligned} \tag{15}$$

(8) Nodes interconnection:

$$\begin{aligned}&\forall i \in \{0,\ldots,n+k-1\},\quad j \in \{\max(i+1;k+1),\ldots,n+k-1\}, \\ &m \in \{0,1\}: C_{i,j,m} \Longleftrightarrow Y_{i,p} = X_{j,m,p}\,. \end{aligned} \tag{16}$$

In simple words, if there is a connection between the node i and the m^{th} input of the node j, these values are equal. The output of the last node $(n+k)$ cannot be connected anywhere.

This is in CNF:

$$\begin{aligned}&\forall i \in \{0,\ldots,n+k-1\},\quad j \in \{\max(i+1;k+1),\ldots,n+k-1\}, \\ &m \in \{0,1\}: (\overline{C_{i,j,m}} \vee Y_{i,p} \vee \overline{X_{j,m,p}}) \wedge (\overline{C_{i,j,m}} \vee \overline{Y_{i,p}} \vee X_{j,m,p})\,. \end{aligned} \tag{17}$$

(9) Negated output edges:

$$\forall w \in \{0,\ldots,o-1\}: ON_{w,p} = OV_{w,p} \oplus ON_w\,. \tag{18}$$

This is in CNF:

$$\begin{aligned}\forall w \in \{0,\ldots,o-1\}: & \\ & (ON_{w,p} \vee OV_{w,p} \vee \overline{ON_w}) \wedge (ON_{w,p} \vee \overline{OV_{w,p}} \vee ON_w) \\ & \wedge (\overline{ON_{w,p}} \vee \overline{OV_{w,p}} \vee \overline{ON_w}) \wedge (\overline{ON_{w,p}} \vee OV_{w,p} \vee ON_w)\,. \end{aligned} \tag{19}$$

(10) Polymorphic output edges:

$$\forall w \in \{0,\ldots,o-1\}: ONP_{w,p} = ON_{w,p} \oplus (OP_w \wedge P_p)\,. \tag{20}$$

This is in CNF:

$$\begin{aligned}\forall w \in \{0,\ldots,o-1\} : &(\overline{OP_w} \vee \overline{ONP_{w,p}} \vee ON_{w,p} \vee P_p)\\ &\wedge (\overline{OP_w} \vee ONP_{w,p} \vee \overline{ON_{w,p}} \vee P_p)\\ &\wedge (\overline{OP_w} \vee ONP_{w,p} \vee ON_{w,p} \vee \overline{P_p})\\ &\wedge (\overline{OP_w} \vee \overline{ONP_{w,p}} \vee \overline{ON_{w,p}} \vee \overline{P_p})\\ &\wedge (OP_w \vee ONP_{w,p} \vee \overline{ON_{w,p}})\\ &\wedge (OP_w \vee \overline{ONP_{w,p}} \vee ON_{w,p})\,. \end{aligned} \tag{21}$$

(11) Outputs:

$$\forall w \in \{0,\ldots,o-1\}, \quad i \in \{0,\ldots,n+k\} : O_{w,i} \Rightarrow (OV_{w,p} = Y_{i,p})\,. \tag{22}$$

This is in CNF:

$$\begin{aligned}\forall w \in \{0,\ldots,o-1\}, \quad i \in \{0,\ldots,n+k\} : &(\overline{O_{w,i}} \vee \overline{OV_{w,p}} \vee Y_{i,p})\\ &\wedge (\overline{O_{w,i}} \vee OV_{w,p} \vee \overline{Y_{i,p}})\,. \end{aligned} \tag{23}$$

(12) Node functions:

$$\forall i \in \{k+1,\ldots,n+k\} : Y_{i,p} = XN_{i,0,p}\langle \mathrm{op}\rangle XN_{i,1,p}\,. \tag{24}$$

The CNF of the operator $\langle \mathrm{op}\rangle$ (node functions) is constructed by deriving the onset and offset of the operator function by simulation. That is, its characteristic function in CNF is obtained. Depending on the function selection (variables $F_{i,v}$), constraints for the outputs ($Y_{i,p}$) and inputs ($XN_{i,0,p}, XN_{i,1,p}$) of nodes are derived, based on this node characteristic function. The resulting SAT instance will contain one clause per minterm of the node function.

For the sake of brevity, we will omit a detailed formal description of the procedure.

Just to give a simple example, if, e.g., $F_{i,0} = 0$ selects that the node i will implement an AND gate, these constraints will be generated:

$$\forall i \in \{k+1,\ldots,n+k\} : F_{i,0} = 0 \Rightarrow (Y_{i,p} = XN_{i,0,p} \cdot XN_{i,1,p})\,. \tag{25}$$

As a result of enumeration of all minterms of the AND function and including the condition $F_{i,0} = 0$, the following CNF is produced:

$$\begin{aligned}\forall i \in \{k+1,\ldots,n+k\} : &(F_{i,0} \vee XN_{i,0,p} \vee XN_{i,1,p} \vee \overline{Y_{i,p}})\\ &\wedge (F_{i,0} \vee XN_{i,0,p} \vee \overline{XN_{i,1,p}} \vee \overline{Y_{i,p}})\\ &\wedge (F_{i,0} \vee \overline{XN_{i,0,p}} \vee XN_{i,1,p} \vee \overline{Y_{i,p}})\\ &\wedge (F_{i,0} \vee \overline{XN_{i,0,p}} \vee \overline{XN_{i,1,p}} \vee Y_{i,p})\,. \end{aligned} \tag{26}$$

(13) The function — input and output values:

$$\forall i \in \{1, \ldots, k\} : Y_{i,p} = \text{forced respective bit value},$$
$$P_p = \text{polymorphic stimulus value for the } p^{\text{th}} \text{ minterm},$$
$$\forall w \in \{0, \ldots, o-1\}: ONP_{w,p} = \text{forced outputs for minterm } p.$$

These values are directly obtained from the truth table of the designed function and the constraints are implemented in the CNF as unit clauses.

Finally, all the clauses stated in the above subsections are concatenated to form a CNF, to produce a SAT instance. A solution of this instance, particularly the values of variables $N_{i,m}, P_{i,m}, C_{i,j,m}, O_{w,i}, ON_w, OP_w$, and $F_{i,v}$, then represents the implementation of the desired DAG.

Note that the number of clauses describing the DAG validity grows linearly with both k and n, but the number of clauses describing the function grows exponentially with k, because of an exponential number of minterms. When combined with an NP-complete SAT-solving repeatedly run in the process, it is clear that this approach is feasible for small k's only. However, it is fully sufficient for some purposes.[5]

4.3. *Exploiting special properties of gates*

Some gates listed in Table 1 exhibit special properties, like symmetry and symmetry to negation (of course, these properties can be generalized to any set of functions with more than two inputs). These properties can be used to prune the search space efficiently. Particularly, additional rules are introduced and SAT clauses are added to prevent ambiguity.

4.3.1. *Symmetric functions*

For symmetric functions, a node with a lower index is always the left child of its parent:

$$\forall i \in \{k+1, \ldots, n+k\}, \quad j \in \{0, \ldots, i-1\}, \quad h \in \{0, \ldots, j\} : C_{j,i,0} \Rightarrow \overline{C_{h,i,1}}\,. \quad (27)$$

This is in CNF:

$$\forall i \in \{k+1, \ldots, n+k\}, \quad j \in \{0, \ldots, i-1\}, \quad h \in \{0, \ldots, j\} : \overline{C_{j,i,0}} \vee \overline{C_{h,i,1}}\,. \quad (28)$$

If all allowed functions are symmetric, the above clauses are added to the CNF without any modification. In case there are nonsymmetric functions allowed, the clauses are extended by identification of the function implementing node i ($F_{i,v}$).

4.3.2. *Symmetry to negation*

The property of a function $a \circ \bar{b} = \bar{a} \circ b = \overline{a \circ b}$ (case of, e.g., a XOR function) allows for a significant reduction of possibilities of placing the negation. In our implementation,

we prohibit using negated edges at both nodes' inputs; if negation is required, it is placed at the output of such a node (which is the input of another node or a PO). If there is a chain of nodes with this property, the negation is placed at the output of this chain.

The following constraints are added to the CNF, for nodes implemented as functions symmetric to negations (which is given by $F_{i,v}$ values):

$$\forall i \in \{k+1,\ldots,n+k\},\quad v \in \{0,\ldots,\lceil \log_2|\mathbf{F}|\rceil - 1\},\quad m \in \{0,1\} : F_{i,v} \Rightarrow \overline{N_{i,m}}\,. \quad (29)$$

This is in CNF:

$$\forall i \in \{k,\ldots,n+k\},\quad v \in \{0,\ldots,\lceil \log_2|\mathbf{F}|\rceil - 1\},\quad m \in \{0,1\} : \overline{F_{i,v}} \vee \overline{N_{i,m}}\,. \quad (30)$$

4.4. *Enumeration*

In order to obtain multiple or even all solutions, i.e., all optimum DAG structures implementing the given function, an All-SAT-solver can simply be used. However, this approach is not practical, since many structurally equivalent solutions with just permuted node indexes would be produced. Therefore, we propose the procedure shown in Fig. 6.

Here, the best n (the minimum number of nodes by which the function can be implemented) with the initial solution is found first. Actually, this procedure is just equal to the original algorithm (Fig. 4). Then the CNF is constrained so that such a solution will not be generated by a consequent SAT-solver run. This is done by

```
Generate_all (truth_table f, int k) {
   // find minimum n first
   n = 1;
   do {
      CNF = Generate_CNF(f, k, n);
      Sol = SAT_Solve(CNF);
      if (Sol.unsat) n++;
   } while (Sol.unsat);

   // enumerate all solutions
   while (!Sol.unsat) {
      All.append(Sol.extract_structure);
      for_all_feasible_permutations P {
         CNF.Constrain(P, Sol);
      }
      Sol = SAT_Solve(CNF);
   }
   return All;
}
```

Fig. 6. All optimal structures generation procedure.

simply adding *blocking clauses* describing the solution, i.e., the disjunction of all variables $N_{i,m}, P_{i,m}, C_{i,j,m}, O_{w,i}, ON_w, OP_w$, and $F_{i,v}$ to the CNF, while the polarities of variables correspond to the variables' values in the solution (e.g., if a variable x value is "1" in the solution, the literal $\bar{x}$ is present in the blocking clause). This is done for *all permutations* of node indexes, which describe a valid DAG (i.e., where conditions from Sec. 4.2 are satisfied).

Then the SAT-solver is invoked for this CNF instance. This procedure is repeated until an unsatisfiable solution is obtained, indicating that no other feasible solutions exist.

As a result, all feasible nonisomorphic solutions are produced.

4.5. *Deriving depth-optimal implementations*

Even though the procedures described above produce optimal implementations in terms of nodes count, they need not be delay-optimal (or depth-optimal). The approach presented in Ref. 36 uses SMT to evaluate the level of each node and optimize the result according to this. Here the level is expressed as an integer variable attached to each node. A SAT-based approach producing delay-optimal solutions is presented in Ref. 35. In principle, a similar approach is adopted; the delays of each node are represented by bitstrings.

For the sake of simplicity, we have opted for a different approach based on the enumeration. The enumeration procedure is run (see Sec. 4.4) and only one depth-optimal solution is selected as a result. Naturally, lower bounds on circuit depth can be imposed. Thus this approach does not involve enumerating all solutions in most cases.

4.6. *Customizable gate costs*

The basic procedure presented in Sec. 4.1 assumed equal costs of all nodes. However, this may not be always desired. Thus, we propose an approach based on PBO, which can assume *any integer* cost of *any* node type. In its basic form, the implementation is simple and straightforward. The CNF clauses from Sec. 4.2 are transformed into PBO inequalities, as described in Sec. 3.4.

Next, the optimization criterion to be minimized is defined. The node functions are primarily encoded in binary (variables $F_{i,v}$). This representation is however not well suitable for the node cost computation in PBO. Thus, we first append its "translation" to the one-hot encoding to the CNF, forming new variables:

- $FH_{i,z}$ denotes the i^{th} node function selector in one hot encoding, $i \in \{k+1, \ldots, n+k\}$, $z \in \{0, \ldots, |\mathbf{F}|-1\}$.

We will not mention the respective CNF clauses performing the translation from $F_{i,v}$ to $FH_{i,z}$, for the sake of brevity.

Then, the optimization criterion to be minimized is defined, by summing the costs:

$$\sum_{i\in\{k+1,\ldots,n+k\}} \left(\sum_{z\in\{0,\ldots,|\mathbf{F}|-1\}} FH_{i,z} \cdot \mathrm{cost}(z) \right). \tag{31}$$

Here the function $\mathrm{cost}(z)$ returns the integer cost of the z^{th} gate type.

A solution of this PBO instance results in an implementation consisted of n gates reducing the total cost.

This approach, however, cannot be directly used in the context of the overall algorithm, see Sec. 4.1. For example, let us assume an implementation using AND and XOR gates, with the XOR cost being 3 and the AND cost 1. Assume the initial solution of $n = 3$ with one XOR gate has been found. Therefore, the total cost is 5 (two AND gates + one XOR gate). However, we are not sure if there is not a cheaper solution comprised of just four AND gates (i.e., having the cost equal to 4).

Therefore, the algorithm from Fig. 4 must be slightly modified, as shown in Fig. 7. The algorithm starts similarly to the original one (Fig. 4); only a PBO-solver is used instead. In this phase, the *initial* solution primarily minimizing the total number of nodes and secondarily the cost is obtained. This solution is recorded as the best solution found so far.

The second phase of the algorithm tries to find a "cheaper" solution consisted of *more* nodes. Increasing n makes sense while it is lower than the cost of the best solution obtained above; in case of equality, the best solution consists of the "cheapest" gates only (the minimum cost is assumed to be 1).

When searching for a better solution having more nodes than the initial one, the set of gates that can be used can be restricted, based on their cost (the `ConstrainCost` procedure in Fig. 7). Also note that the PBO-solver needs not be employed in all cases and a (faster) SAT-solver can be used instead. This happens when there is no freedom left in the choice of gates. In practice, the PBO-solver is called only once or twice at the beginning of the process, based on the gates' cost span.

Let us recall the above example. After the initial solution is obtained ($n = 3$, cost = 5, two ANDs, one XOR), n is increased to four in the next phase. In order to obtain a solution with smaller cost (i.e., four), no XOR gate can be used, and thus there is also no freedom of gates choice. Therefore, the CNF is restricted so that only AND gates are allowed and SAT is solved instead of PBO.

The enumeration and delay optimizing versions of the algorithm (see Fig. 6) must be modified in a similar way. Thus, starting with the n value obtained from the algorithm in Fig. 7, n must be increased up to the `Best_Sol.cost` value to find all optimum solutions. The difference from the algorithm in Fig. 7 is the addition of blocking clauses after each solution found, and repetition of the main loop, while satisfiable solutions are generated, similarly to Fig. 6.

```
Generate_structure (truth_table f, int k) {
   // find initial solution
   n = 1;
   do {
      CNF = Generate_CNF(f, k, n);
      PBO = CNF.ToPBO();
      Sol = PBO_Solve(PBO);
      if (Sol.unsat) n++;
   } while (Sol.unsat);
   Best_Sol = Sol;

   // try to find better solution
   while (n < Best_Sol.cost) {
      n++; // increase nodes count
      CNF = Generate_CNF(f, k, n);
      ConstrainCost(best_cost - n);
      if (best_cost - n > 1) {
         PBO = CNF.ToPBO();
         Sol = PBO_Solve(PBO);
      } else Sol = SAT_Solve(CNF);
      if (Sol.cost < Best_Sol.cost) {
         Best_Sol = Sol;
      }
   }
   return Best_Sol.extract_structure();
}
```

Fig. 7. The optimal structure generation procedure using PBO.

5. Experimental Results

5.1. *Experimental setup*

The experimental results are demonstrated in this section. MiniSAT[37] has been used as a SAT-solver, MiniSAT+ as a PBO-solver.[38] All the computations were performed on a computer cloud with Intel Xeon E5-2630v3 2.40-GHz CPUs and 128-GB RAM.

5.2. *Synthesis of replacement structures for rewriting*

As it was stated above, the *rewriting*-based logic optimization[1,5] is one of the applications where optimum implementations of functions are required. Here, a logic network is optimized by repeatedly replacing its k-input subgraphs by their optimum implementations. For this purpose, we need optimum implementations of all k-input functions, or, better, their representative NPN-equivalence classes.[2–4]

In this experiment, we have synthesized *all* nonisomorphic implementations of *all* representatives of the 222 NPN-equivalence classes of four-input functions.[2] As

Table 2. Four-input rewriting structures statistics.

	Nodes		XORs		Count		
AND:XOR	Max.	Avg.	Max.	Avg.	Max.	Avg.	Total
1:1	7	6.60	5	2.78	7,401	144.86	32,160
1:2	8	6.38	3	1.41	2,436	42.58	9,453
1:3	10	8.02	3	0.23	3,056	95.45	21,190

the implementation basis, the set {AND, XOR} was chosen. No polymorphism was used in this experiment. The AND node cost was set to 1, while the XOR cost varied from 1 to 3.

The summary results are shown in Table 2, where the total numbers of nodes, XORs, and the counts of produced structures are shown.

We can see that with an increasing XOR cost, the number of XOR gates present in the implementation shrinks and the total number of nodes increases.

The runtimes are not present since this experiment was rather time-consuming; there are functions that have far too many solutions (see the 1:1 ratio), making the enumeration to run very slow, because of excessive size of blocking clauses. However, computing a single replacement structure typically takes negligible time.

5.3. *Synthesis of replacement structures for polymorphic rewriting*

The idea of rewriting can be extended to *polymorphic rewriting*. Let us assume that each subgraph may implement two different functions, based on the value of the polymorphic stimulus. Thus, we need optimum implementations of all *pairs* of functions. Since there are 222 NPN-equivalence classes of four-input functions, producing all pairs may be prohibitive. However, for three-input functions it is feasible, as there are only 14 such classes.[2] Thus, there are $14^2 = 192$ such functions. Subtracting 14 nonpolymorphic functions, we need 182 different optimum implementations.

We have generated these implementations, with the gate basis comprised of AND and XOR gates. Both the AND and XOR costs were set to 1. The statistics are shown in Table 3.

We can see that implementations of most of the functions were generated very quickly, in an almost unmeasurable time. Generation of the whole set of 182 functions took less than 1 min. Therefore, we can credibly conclude that the presented method can be efficiently used for online computation of rewriting structures for polymorphic rewriting.

5.4. *Synthesis of small polymorphic circuits*

In this subsection, we present synthesis results for several smallest circuits from the MCNC[51] and ITC'99[52] benchmark sets, plus generic adders. Combinational parts of sequential circuits were extracted. These circuits were collapsed to a PLA by ABC[53] to obtain a truth table. For the purpose of this experiment, we have defined a

Table 3. Polymorphic rewriting structures statistics for three-input functions.

	Min.	Max.	Avg.	Sum
Gates	1	6	4.06	739
XORs	0	3	1.10	200
Polymorphic edges	1	8	4.19	762
Runtime (s)	< 0.01	301	1.66	41.58

scenario where the first circuit input is defined as the polymorphic stimulus, while the other inputs remain the PIs. Such a scenario is not that unrealistic — one may imagine a technology, where the function of multiple gates is influenced just by one signal. The Si-NW technology is such an example.[54]

Optimum implementations of the example circuits have been synthesized, without polymorphism being used, and compared to their respective optimum polymorphic designs. The results are shown in Table 4 for AND-nodes-based circuits. Similar results are presented in Table 5, with the nodes set extended by an XOR gate (with its cost set equal to the AND gate cost).

After the circuit name, the numbers of its inputs and outputs (k, o) are given. Then, results of standard optimum synthesis[11] are shown, in terms of the number of nodes (Nd.) and levels (Lev.), followed by the results of polymorphic

Table 4. Synthesis results — only AND nodes used.

			Standard logic			Polymorphic logic			
Name	k	o	Nd.	Lev.	Time	Nd.	P. edges	Lev.	Time
01-adder	3	2	7	4	0.20	3	6	2	0.04
02-adder	5	3	—	—	—	10	6	6	7.01
ITC_b01	6	7	—	—	—	12	12	3	48.33
ITC_b02	4	4	—	—	—	9	12	4	1.62
ITC_b06	10	14	10	4	17,429.40	5	9	2	6,033.40
b1	3	4	6	3	0.23	1	3	1	0.02
c17	5	2	6	3	1.91	5	3	3	1.06
clpl	11	5	10	2	33,596.10	10	3	2	33,199.00
cm82a	5	3	—	—	—	10	11	5	5.68
daio	5	6	10	4	10.63	9	13	3	6.46
dc1	4	7	—	—	—	13	17	3	5.85
lion	4	3	9	4	1.71	5	6	2	0.30
majority	5	1	8	5	3.78	5	6	4	0.61
mc	5	7	—	—	—	9	7	3	7.94
newcwp	4	5	—	—	—	9	5	3	1.94
newtag	8	1	9	6	276.02	8	12	4	181.68
s27	7	4	7	4	42.10	7	6	3	48.98
t	5	2	6	3	1.96	6	5	3	1.33
wim	4	7	—	—	—	12	12	4	4.71
xor5	5	1	—	—	—	9	10	4	4.04
Sum			**88**	**42**	**51,364.04**	**64 (27%)**	**72**	**29 (30%)**	**39,472.88(23%)**

Table 5. Synthesis results–AND and XOR nodes used.

	Standard logic				Polymorphic logic				
Name	Nd.	XORs	Lev.	Time	Nd.	XORs	P. edges	Lev.	Time
01-adder	5	2	2	0.10	2	1	6	1	0.04
02-adder	—	—	—	—	7	4	13	4	2.96
ITC_b01	—	—	—	—	10	3	14	5	51.56
ITC_b02	—	—	—	—	9	1	10	5	4.03
ITC_b06	8	3	4	12,373.40	5	0	9	2	6,475.23
b1	3	2	2	0.10	1	0	3	1	0.02
c17	6	0	3	2.03	5	0	3	3	1.02
clpl	10	0	2	52,105.60	10	0	2	2	51,541.30
cm82a	10	6	6	8.20	7	4	12	3	2.81
daio	7	2	3	5.08	7	3	9	3	4.01
dc1	—	—	—	—	12	2	17	3	6.33
lion	9	2	6	1.85	5	0	4	2	0.43
majority	8	3	4	3.80	5	0	4	4	0.92
mc	10	5	4	14.89	9	1	6	3	6.98
newcwp	8	5	3	1.70	6	3	8	3	0.80
newtag	—	—	—	—	8	1	8	6	201.56
s27	7	0	4	44.10	7	0	5	3	46.08
t	6	0	2	2.10	6	0	6	2	1.73
wim	—	—	—	—	11	2	14	3	3.86
xor5	4	4	3	0.64	3	3	5	3	0.29
Sum	**101**	**34**	**48**	**64,563.59**	**78 (23%)**	**15** (56%)	**82**	**35 (27%)**	**58,081.65(10%)**

implementations. As mentioned above, the first circuit input was always selected as polymorphic stimulus, the "P. edges" column gives the number of polymorphic edges in the implementation.

Results of only 20 circuits are shown in the tables, with summary values given in the last rows, together with the overall percentage reductions compared to the standard synthesis process. For some circuits we were not able to obtain implementations using standard (nonpolymorphic) logic, thus the summary results are computed from the complete records only.

It is possible to see that the polymorphic implementations exhibit smaller area and number of levels in most of the cases, assuming that the polymorphic edges are "for free". Even though this observation is quite obvious, since the notion of polymorphism actually allows implementing implicit XOR gates (see Sec. 3.2), the purpose of the experiments was to illustrate *how much* area can be saved by means of using the polymorphic electronics paradigm.

6. Conclusions

A SAT-based method to generate optimum polymorphic circuits described by a DAG was presented. The polymorphic behavior is implemented by the introduction of polymorphic edges into the DAG.

The method is general, in the sense that any set of two-input gates can be used as the circuit building blocks (DAG nodes). These gates can be assigned an arbitrary cost (size), allowing to produce circuits minimizing this cost.

Since the complexity of the problem solved — the optimum circuit — is immense (as it is a Σ_2^P-complete problem), the method can be applied to functions having typically up to 10 inputs. However, the method can still serve as a means of obtaining lower bounds of complexity of polymorphic circuits.

The feasibility of the method has been illustrated by experimental results. We have compared "standard" optimum implementations of several benchmark circuits with their polymorphic counterparts. A scenario, where one circuit input serves as a polymorphic stimulus, was used. As a result, polymorphic implementations exhibit an average 27% improvement in the number of gates, when XOR gates are not allowed, and an average improvement of 23% for AND–XOR logic. However, polymorphism is considered to be available "for free" in this scenario.

Presenting this experimental comparison, however, was not the main purpose of this paper, since it is unrealistic unless particular technology is targeted. Instead, the objective was to present the method itself, in its most general form. Then, it can be used, e.g., as a part of more complex synthesis algorithms applied to specific target technologies, where different design primitives with different costs are to be used.

One of the promising application areas is the generation of optimum implementations of functions with a limited number of inputs, to be used in general logic optimization processes, like *rewriting*.[1,5] Here, optimum implementations of "small" functions are used to replace their functional equivalents in a circuit to be optimized, to reduce its size. For this purpose, all implementations of all 222 NPN-equivalence classes of four-input functions were generated for standard AND–XOR logic and the statistics were measured, as a part of the experiments. Next, representatives of 182 NPN-equivalence classes of pairs of three-input polymorphic functions were generated and the runtime was measured. Very good applicability of the approach to online generation of replacement structures was illustrated this way.

Acknowledgments

This work was 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). Another support for this work has been gratefully provided by the Grant FIT-S-17-3994 of Brno University of Technology, "Advanced parallel and embedded computer systems" (2017-2019). Access to computing and storage facilities owned by parties and projects contributing to the National Grid Infrastructure MetaCentrum, provided under the program "Projects of Large Research, Development, and Innovations Infrastructures" (CESNET LM2015042), is greatly appreciated.

The authors acknowledge the support of the OP VVV MEYS funded Project CZ.02.1.01/0.0/0.0/16_019/0000765 "Research Center for Informatics".

References

1. A. Mishchenko, S. Chatterjee and R. K. Brayton, DAG-aware AIG rewriting: A fresh look at combinational logic synthesis, *Proc. 43th Design Automation Conf.* (2006), pp. 532–535.
2. M. A. Harrison, The number of equivalence classes of Boolean functions under groups containing negation, *IEEE Trans. Electron. Comput.* **EC-12** (1963) 559–561.
3. S. Muroga, *Logic Design and Switching Theory* (John Wiley & Sons, Ltd., 1979).
4. J. N. Culliney, M. H. Young, T. Nakagawa and S. Muroga, Results of the synthesis of optimal networks of AND and OR gates for four variable switching functions, *IEEE Trans. Comput.* **28** (1979) 76–85.
5. I. Halecek, P. Fiser and J. Schmidt, Towards AND/XOR balanced synthesis: Logic circuits rewriting with XOR, *Microelectron. Reliab.* **81** (2018) 274–286.
6. E. Boros and P. L. Hammer, Pseudo-boolean optimization, *Discrete Appl. Math.* **123** (2002) 155–225.
7. C. Umans, The minimum equivalent DNF problem and shortest implicants, *J. Comput. Syst. Sci.* **63** (2001) 597–611.
8. D. Buchfuhrer and C. Umans, The complexity of Boolean formula minimization, *J. Comput. Syst. Sci.* **77** (2011) 142–153.
9. C. Barrett, R. Sebastiani, S. Seshia and C. Tinelli, Satisfiability modulo theories, *Handbook of Satisfiability*, Frontiers in Artificial Intelligence and Applications, Vol. 185 (IOS Press, 2009), pp. 825–885.
10. P. Fiser and V. Simek, Optimum polymorphic circuits synthesis method, *Proc. 13th IEEE Int. Conf. Design and Technology of Integrated Systems in Nanoscale Era (DTIS)* (2018), p. 6.
11. P. Fiser, I. Halecek and J. Schmidt, SAT-based generation of optimum function implementations with XOR gates, *Proc. 20th Euromicro Conf. Digital Systems Design (DSD)* (2017), pp. 163–170.
12. I. L. Markov, Limits on fundamental limits to computation, *Nature* **512** (2014) 147–154.
13. A. Stoica, Polymorphic electronics: A novel type of circuits with multiple functionality, NASA New Technology Report No. NPO-21213, NASA (2000).
14. A. Stoica, EHW approach to temperature compensation of electronics, NASA Technical Briefs No. NPO-21146, NASA (2004).
15. A. Stoica, R. S. Zebulum and D. Keymeulen and J. Lohn, On polymorphic circuits and their design using evolutionary algorithms, *Proc. 20th IASTED Int. Conf. Applied Informatics* (2002), pp. 1–6.
16. A. Stoica and R. S. Zebulum, Multifunctional logic gate controlled by temperature, NASA Tech Briefs No. NPO-30795, NASA's Jet propulsion Laboratory, Pasadena, CA (2005).
17. A. Stoica and R. S. Zebulum, Polymorphic electronic circuits, NASA Tech Briefs No. NPO-21213, NASA's Jet Propulsion Laboratory, Pasadena, CA (2004).
18. Z. Gajda and L. Sekanina, On evolutionary synthesis of compact polymorphic combinational circuits, *J. Mult.- Valued Logic Soft Comput.* **17** (2011) 607–631.
19. Z. Gajda, Evolutionary approach to synthesis and optimization of ordinary and polymorphic circuits, Ph.D. thesis, Department of Computer Systems, Faculity of Information Technology, Brno University of Technology (2011), p. 92.

20. R. Tesar, V. Simek, R. Ruzicka and A. Crha, Design of polymorphic operators for efficient synthesis of multifunctional circuits, *J. Comput. Commun.* **4** (2016) 151–159.
21. A. Crha, R. Ruzicka and V. Simek, Synthesis methodology of polymorphic circuits using polymorphic NAND/NOR gates, *Int. Conf. Mathematical/Analytical Modelling and Computer Simulation* (2015), pp. 612–617.
22. E. L. Lawler, An approach to multilevel boolean minimization, *J. ACM* **11** (1964) 283–295.
23. R. M. Karp, F. E. McFarlin, J. P. Roth and J. R. Wilts, A computer program for the synthesis of combinational switching circuits, *Proc. 2nd Annu. Symp. Switching Circuit Theory and Logical Design* (1961), pp. 182–194.
24. J. P. Roth and R. M. Karp, Minimization over Boolean graphs, *IBM J. Res. Develop.* **6** (1962) 227–238.
25. P. R. Schneider and D. L. Dietmeyer, An algorithm for synthesis of multiple-output combinational logic, *IEEE Trans. Comput.* **C-17** (1968) 117–128.
26. E. S. Davidson, An algorithm for NAND decomposition under network constraints, *IEEE Trans. Comput.* **18** (1969) 1098–1109.
27. T. Nakagawa, A branch-and-bound algorithm for optimal AND-OR networks (the algorithm description), Report No. UIUCDCS-R-71-462. Department of Computer Science, University of Illinois Urbana-Champaign (1971).
28. E. A. Ernst, Optimal combinational multi-level logic synthesis, Ph.D. thesis, The University of Michigan (2009).
29. R. Drechsler and W. Günther, Exact circuit synthesis, *Proc. Int. Workshop Logic & Synthesis (IWLS)* (1998).
30. C. R. Baugh, T. Ibaraki and S. Muroga, Technical note: Results in using Gomory's all-integer integer algorithm to design optimum logic networks, *Oper. Res.* **19** (1971) 1090–1096.
31. S. Muroga and T. Ibaraki, Design of optimal switching networks by integer programming, *IEEE Trans. Comput.* **21** (1972) 573–582.
32. S. Muroga and H. C. Lai, Minimization of logic networks under a generalized cost function, *IEEE Trans. Comput.* **25** (1976) 893–907.
33. A. Kojevnikov, A. S. Kulikov and G. Yaroslavtsev, Finding efficient circuits using SAT-solvers, *Proc. Int. Conf. Theory and Applications of Satisfiability Testing* (2009), pp. 32–44.
34. I. Halecek, P. Fiser and J. Schmidt, Are XORs in logic synthesis really necessary? *Proc. IEEE 20th Int. Symp. Design and Diagnostics of Electronic Circuits & Systems (DDECS)* (2017), pp. 138–143.
35. M. Soeken *et al.*, Practical exact synthesis, *Proc. Design, Automation & Test in Europe Conf. & Exhibition (DATE)* (2018), pp. 309–314.
36. M. Soeken *et al.*, Exact synthesis of majority-inverter graphs and its applications, *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* **36** (2017) 1842–1855.
37. N. Een and N. Sorensson, An extensible SAT-solver, *SAT 2003: Theory and Application of Satisfiability Testing*, Lecture Notes in Computer, Science, Vol. 2919 (Springer-Verlag, 2004), pp. 333–336.
38. N. Een and N. Soerensson, Translating pseudo-Boolean constraints into SAT, *J. Satisf. Boolean Model. Comput.* **2** (2006) 1–26.
39. S. B. Akers, Binary decision diagrams, *IEEE Trans. Comput.* **27** (1978) 509–516.
40. M. Fujita, P. C. McGeer and J. C. Yang, Multi-terminal binary decision diagrams: An efficient data structure for matrix representation, *Form. Methods Syst. Des.* **10** (1997) 149–169.

41. G. D. Hachtel and F. Somenzi, *Logic Synthesis and Verification Algorithms* (Kluwer Academic, Boston, 1996).
42. P. Bjesse and A. Borlv, DAG-aware circuit compression for formal verification, *Proc. IEEE/ACM Int. Conf. Computer-Aided Design* (2004), pp. 42–49.
43. P. A. Abdullah, P. Bjesse and N. Een, Symbolic reachability analysis based on SAT-solvers, *Proc. 9th Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems* (2000).
44. A. Kuehlmann, V. Paruthi, F. Krohm and M. Ganai, Robust Boolean reasoning for equivalence checking and functional property verification, *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* **21** (2001) 1377–1394.
45. A. Crha, V. Simek and R. Ruzicka, Towards novel format for representation of polymorphic circuits, *Proc. 13th Int. Conf. Design & Technology of Integrated Systems in Nanoscale Era (DTIS)* (2018), pp. 1–2.
46. M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman & Co. (New York, 1990), p. 338.
47. F. Aloul, A. Ramani, I. Markov and K. Sakallah, PBS, a backtrack search pseudo-Boolean solver, *Proc. Fifth Int. Symp. Theory and Applications of Satisfiability Testing (SAT)* (2002).
48. V. M. Manquinho and O. Roussel, The first evaluation of pseudo-Boolean solvers, *J. Satisf. Boolean Model. Comput.* **2** (2006) 103–143.
49. H. M. Sheini and K. A. Sakallah, Pueblo: A hybrid pseudo-Boolean SAT solver, *J. Satisf. Boolean Model. Comput.* **2** (2006) 165–189.
50. O. Bailleux, Y. Boufkhad and O. Roussel, A translation of pseudo Boolean constraints to SAT, *J. Satisf. Boolean Model. Comput.* **2** (2006) 191–200.
51. S. Yang, Logic synthesis and optimization benchmarks user guide: Version 3.0, MCNC Technical Report, MCNC, NC, USA (1991).
52. F. Corno, M. S. Reorda and G. Squillero, RT-level ITC'99 benchmarks and first ATPG results, *Proc. IEEE Design Test of Computers* (2000).
53. Berkeley Logic Synthesis and Verification Group, ABC: A system for sequential synthesis and verification (2018), http://www.eecs.berkeley.edu/alanmi/abc/.
54. L. Amaru *et al.*, New logic synthesis as nanotechnology enabler, *Proc. IEEE* **103** (2015) 2168–2195.