Practical Exact Synthesis

(Executive Session Paper)

Mathias Soeken[1] Winston Haaswijk[1] Eleonora Testa[1] Alan Mishchenko[2] Luca G. Amarù[3]
Robert K. Brayton[2] Giovanni De Micheli[1]
[1]Integrated Systems Laboratory, EPFL, Switzerland
[2]EECS, UC Berkeley, CA, USA
[3]Synopsys Inc., CA, USA

***Abstract*—In this paper, we discuss recent advances in exact synthesis, considering both their efficient implementation and various applications in which they can be employed. We emphasize on solving exact synthesis through Boolean satisfiability (SAT) encodings. Different SAT encodings for exact synthesis are compared, and examined the applications to multi-level logic synthesis, in both area and depth optimization. Another application of SAT based exact synthesis is optimization under many constraints. These constraints can, e.g., be a fixed fanout or delay constraints. Finally, we end our discussion by proposing directions for future research in exact synthesis.**

I. Introduction

Exact synthesis is the problem of finding the optimum logic representation for a given Boolean function with respect to some cost criteria. Specific instances of the problem are finding the sum-of-product (SOP) representation using the smallest number of implicants, or a 2-input gate level logic network with the fewest number of gates. For theoretical purposes, one often considers the whole set of Boolean functions for a fixed number of variables n. This allows us to derive lower bounds on the complexity of logic representations. It is known that all 4-variable Boolean functions can be represented using SOPs with at most 8 implicants [1]. Also, all 5-variable Boolean functions can be represented using 2-input gate-level networks with at most 12 gates [2]. Since the number of Boolean functions grows double-exponentially, it is hard to derive such results for larger number of variables.

In practical applications, we are not interested in finding logic representations for *all* Boolean functions, but instead only in a small subset of Boolean functions that are found during synthesis and optimization [3]. Recent advances in the implementation of exact synthesis algorithms—particularly SAT-based implementations—significantly improved their performance and thereby widened the applications in which they are employed. This makes it viable to consider SAT-based exact synthesis an essential engine in modern practical logic synthesis applications.

This paper summarizes the state-of-the-art and reviews the most important algorithmic details. The next section introduces preliminaries to ease the formal notation in the remaining sections. Section III illustrates the main SAT-based formulation and algorithms for the general case. Section IV discusses how to consider delay constraints, whereas Section V discusses how to include arbitrary constraints for logic synthesis applications with many and complex constraints. The paper concludes by mentioning future challenges in Section VI.

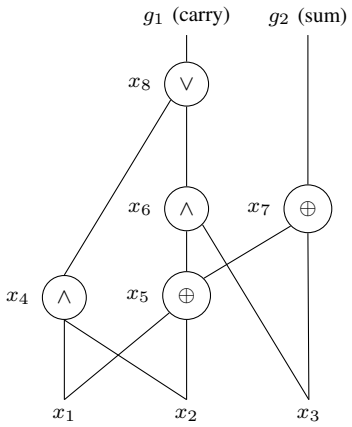

Fig. 1: Illustration of a Boolean logic network for a full adder. The steps are labeled with their corresponding Boolean operators.

II. Preliminaries

This section introduces definitions and notations on Boolean logic networks and Boolean satisfiability solving.

A. Boolean logic networks

Combinatorial Boolean logic networks are directed acyclic graphs. A vertex with in-degree 0 is a primary input and a vertex with out-degree 0 is a primary output. All other vertices correspond to Boolean logic gates that can represent an arbitrary k-variable Boolean function, where k is the in-degree of the vertex. In practice, we often restrict the syntactical expressiveness of a logic network by (i) restricting the in-degree (number of inputs) of a logic gate, or (ii) restricting which functions can be realized by the logic gates.

In our ongoing discussions on exact synthesis we will consider logic networks in which each logic gate has two inputs, and can realize any of the 16 2-input Boolean functions. We make use of a formal notation inspired by Boolean chains in [2], since they allow for an easier description of the exact synthesis formulations in the following sections. Formally, a (two-input gate) Boolean network for n inputs $x_1, \ldots, x_n$ is a sequence of gates $(x_{n+1}, \ldots, x_{n+r})$ with

$$x_i = x_{j(i)} \circ_i x_{k(i)}, \qquad \text{for } n+1 \leq i \leq n+r. \tag{1}$$

That is, each gate combines two previous gates or inputs with $j(i) < k(i) < i$ using $\circ_i$, which is one of the 2-input Boolean functions. For single-output functions, the last gate

x_{n+r} is considered the network's output. For multi-output networks, each gate could potentially be an output. We call a single-output function f *normal*, if $f(0,\ldots,0) = 0$. A multi-output function is normal, if all of its component functions are normal. A Boolean network represents normal functions if all of its gate functions are normal.

Example 1: Fig. 1 shows an example network for a full adder with three inputs consisting of five gates:

$$x_4 = x_1 \wedge x_2, \quad x_5 = x_1 \oplus x_2, \quad x_6 = x_3 \wedge x_5,$$
$$x_7 = x_3 \oplus x_5, \quad x_8 = x_4 \vee x_6$$

The network has two outputs $g_1 = x_8$ for the carry and $g_2 = x_7$ for the sum.

We refer to the number of gates r as the *size* of the network. The *logic depth* is the length of the longest path between primary inputs and primary outputs. The length of a path is measured in terms of the number of gates on the path.

Example 2: The delay of the logic network in the example is 3, as three gates are on the path from inputs x_1 and x_2 to output g_1.

We can assume different input arrival times for the inputs of the logic network. This is especially important if we consider subnetworks in the context of a larger network. They influence the overall delay of the network. We refer to δ_i as the input arrival time of input x_i. When computing the delay, the input arrival time needs to be added to the length of the path.

Example 3: Assume $\delta_1 = 0$, $\delta_2 = 0$, and $\delta_3 = 2$. Then, the delay of the network is 4, since now the path from x_3 to g_1 has length 2.

B. Boolean satisfiability

Given a Boolean formula $f(x_1,\ldots,x_n)$, the *Boolean satisfiability* problem (or SAT problem) asks whether there exists an assignment to the variables $x_1,\ldots,x_n$ such that f evaluates to true. If this is the case, such an assignment is called a *satisfying assignment* and f is called *satisfiable*. Otherwise, f is *unsatisfiable*. SAT solvers [4], [5] are software programs that receive a Boolean formula f, typically represented in conjunctive normal form (CNF), and return a satisfying assignment, if and only if f is satisfiable. A CNF is a conjunction of clauses, a clause is a disjunction of literals, and a literal is a variable in regular or complemented form. Since in the remainder we use CNFs only to describe the input to a SAT solver, we also refer to them as SAT formulas.

III. SAT-based Exact Synthesis

Given an m-tuple of m functions over n variables

$$(f_1(x_1,\ldots,x_n),\ldots,f_m(x_1,\ldots,x_n))$$

we can formulate the exact synthesis of these functions as a sequence of decision problems $P_0, P_1, P_2, \ldots$. Problem P_r corresponds to the question: "Can functions $f_1,\ldots,f_m$ be computed by an r-step Boolean logic network?". Each instance P_r can be described by a SAT formula.[1] In this section we describe what such a formula looks like, how additional constraints can speed up the synthesis process, as well as some experimental results. We attribute the SAT formulation described here to Kojevnikov *et al.*, Eén, and Knuth [6], [7], [5]. Knuth [5] improved previous approaches, by restricting synthesis to normal Boolean functions.

[1] In practice, P_0 is often handled as a trivial special case, since it means that all $f_1,\ldots f_m$ are constants or variable projections.

A. Variables

We first define the variables used in the SAT formulation. For $1 \le h \le m$, $n < i \le n + r$, and $0 < t < 2^n$, define the following:

$$x_{it} : \text{t}^{\text{th}} \text{ bit of } x_i\text{'s truth table}$$
$$g_{ih} : [f_h = x_i]$$
$$s_{ijk} : [x_i = x_j \circ_i x_k] \text{ for } 1 \le j < k < i$$
$$f_{ipq} : \circ_i(p,q) \text{ for } 0 \le p,q \le 1, p+q > 0$$

The variables x_{it} correspond to the value (at row t) of the global truth table for step x_i. The g_{ih} variables determine which outputs point to which steps. Thus, if g_{ih} is true, it means that function f_h is computed by step i. The s_{ijk} variables determine, for each step i, the inputs j and k. Also known as *selection variables*, their assignments control the underlying DAG structure of the Boolean network. The f_{ipq} encode for all steps i what the corresponding Boolean operator is. Since we synthesize normal logic networks, we do not need to consider row 0 of the gate's truth tables and require only $2^n - 1$ truth table indices t. Also $p + q > 0$, since the local function describing a gate's operation does not need to be specified for the case $p = q = 0$.

B. Constraints

We now constrain the variables by a set of clauses which ensures that the network computes the correct functions. With the addition of these clauses, the SAT formula is satisfiable *if and only if* the given functions can be computed by an r-step logic network. For $0 \le a,b,c \le 1$ and $1 \le j < k < i$, the main clauses are:

$$((s_{ijk} \wedge (x_{it} \oplus \bar{a}) \wedge (x_{jt} \oplus \bar{b}) \wedge (x_{kt} \oplus \bar{c})) \rightarrow (f_{ibc} \oplus \bar{a}))$$

In other words: if step i has inputs j and k, and the t$^{\text{th}}$ bit of x_i is a, and the t$^{\text{th}}$ bit of x_j is b, and the t$^{\text{th}}$ bit of x_k is c, then we must have $\circ_i(b,c) = a$. We can rewrite these constraints to CNF:

$$(\bar{s}_{ijk} \vee (x_{it} \oplus a) \vee (x_{jt} \oplus b) \vee (x_{kt} \oplus c) \vee (f_{ibc} \oplus \bar{a}))$$

Here, a, b, and c are constants used to set the proper variable polarities. In fact, these constraints may be simplified in several cases. When $b = c = 0$, the final term encodes f_{i00}. If $a = 0$ this is trivially true, due to the normality of the network. Hence, in that case the entire clause may be omitted. If $a = 1$ the final literal is omitted from the clause. Similarly, x_{jt} and x_{kt} are constants if $j \le n$ or $k \le n$, and the appropriate simplifications can be made.

Next, let $(t_1,\ldots,t_n)_2 = t$ be the binary encoding of t, such that t_i refers to the i^{th} bit of t. In order to fix the proper output values, we add the clauses $(\bar{g}_{hi} \vee \bar{x}_{it})$ or $(\bar{g}_{hi} \vee x_{it})$ depending on the value $f_h(t_1,\ldots,t_n)$. Finally, we add the clauses $\bigvee_{i=n+1}^{n+r} g_{hi}$ and $\bigvee_{1 \le j < k < i} s_{ijk}$, so that every output h points to a step in the network and to ensure that every gate i has two inputs.

Example 4: Recall the Boolean logic network from Fig. 1. Let us consider a variable assignment that would synthesize it. It has 5 steps, so the corresponding decision problem is P_5 and $r = 5$. Further, it has 3 inputs and 2 outputs. Hence, indices i and t range from 4 to 5 and from 1 to 7, respectively.

$$\begin{aligned} t &= 7\ 6\ 5\ 4\ 3\ 2\ 1 \\ x_{4t} &= 1\ 0\ 0\ 0\ 1\ 0\ 0 \\ x_{5t} &= 0\ 1\ 1\ 0\ 0\ 1\ 1 \\ x_{6t} &= 0\ 1\ 1\ 0\ 0\ 0\ 0 \\ x_{7t} &= 1\ 0\ 0\ 1\ 0\ 1\ 1 \\ x_{8t} &= 1\ 1\ 1\ 0\ 1\ 0\ 0 \end{aligned}$$

There are two outputs, each of which can point to 5 steps, making for a total of 10 g_{hi} variables. In this case we have $g_{17} = 1$, $g_{28} = 1$, and $g_{hi} = 0$ for all other g_{hi}.

From the DAG structure of the network, we can see that $s_{412} = 1$, $s_{512} = 1$, $s_{635} = 1$, $s_{735} = 1$, and $s_{846} = 1$. All other s_{ijk} are zero.

Finally, the Boolean operators for the different steps are assigned the following values:

$$\begin{aligned} (p,q) &= (1,1)\ (0,1)\ (1,0) \\ f_{4pq} &= \ 1 \quad\ 0 \quad\ 0 \\ f_{5pq} &= \ 0 \quad\ 1 \quad\ 1 \\ f_{6pq} &= \ 1 \quad\ 0 \quad\ 0 \\ f_{7pq} &= \ 0 \quad\ 1 \quad\ 1 \\ f_{8pq} &= \ 1 \quad\ 1 \quad\ 1 \end{aligned}$$

Additional clauses: The above clauses are the minimum ones necessary to ensure that a valid logic network is found. However, we may add additional constraints to boost synthesis speed, such as clauses to force a colexicographic order on the steps. We refer the reader to [5] for the details.

C. Alternative selection variable scheme

We can create equivalent SAT formulas for P_r using an alternative selection variable scheme. Instead of creating variables s_{ijk}, we can create s_{ij} for $n < i \leq n + r$ and $j < i$. This reduces the number of selection variables from

$$\sum_{i=n+1}^{n+r} \binom{i}{2} = \frac{1}{6}(3n^2 + 3nr + r^2 - 1)$$

to

$$\sum_{i=n+1}^{n+r} (i-1) = \frac{1}{2}(2n + r - 1).$$

Thus, it reduces the number of selection variables from polynomial to linear in n and r. On the other hand, it increases the complexity of clauses. The main clauses are now of the form:

$$((s_{ij} \wedge s_{ik} \wedge (x_{it} \oplus \bar{a}) \wedge (x_{jt} \oplus \bar{b}) \wedge (x_{kt} \oplus \bar{c})) \rightarrow (f_{ibc} \oplus \bar{a})).$$

The other clauses must be adapted analogously. Thus, there is a trade-off between the number of variables, and the complexity of clauses. It is not obvious which scheme is superior, and it may vary due to implementations details and with constraints from different problem domains. This selection variable scheme was used in [8], [9]. A similar scheme was used by Kojevnikov [6].

D. Algorithms

Now that we know how to create the SAT formula for P_r, we can use that to construct an exact synthesis algorithm. The pseudocode can be found in Algorithm 1.

Algorithm 1 A simple SAT based exact synthesis algorithm.

```
function SYNTHESIZE(f_1(x_1,...,x_n),...,f_m(x_1,...x_n))
    r ← 1
    s ← get_sat_solver()
    while true do
        reset_sat_solver(s)
        create_vars(P_r)
        create_clauses(P_r)
        if solve(s) = SAT then
            return extract_network(s)
        end if
        r ← r + 1
    end while
end function
```

Given a sequence of functions, the algorithm clearly finds a Boolean logic network of minimal size. Additional functionality can be added to this basic algorithm for different purposes. For example, in practice we often want to limit the runtime of the algorithm. We can do this by using timeouts or SAT conflict limits. Other variants of the algorithm use incremental SAT in a CEGAR loop [10], which can make synthesis faster for some functions.

E. Experimental results

Table I shows experimental results for the synthesis of two sets of functions, using the algorithm described in this section. We have used similar function sets throughout all sections in the paper for an easier comparison. The end of this section comments on application to large functions in practical logic synthesis applications. The set NPN4 consists of all 222 4-input NPN classes. The set FDSD6 consists of 1000 fully disjoint support set (DSD) decomposable functions [11]. Both sets of functions can be fully synthesized in less than 70 seconds. On average, all functions are synthesized in (much) less than a second. Interestingly, the 6-input DSD functions have a lower average runtime than the 4-input NPN classes. This is due to the fact that it is the complexity of Boolean networks which drives runtime, rather than the number of inputs. The complexity of Boolean networks is also known as *combinational complexity* [2]. Hence, the runtime difference is due to the fact that the 4-input NPN classes contain some functions of relatively high complexity. Generally, we expect random functions to be of high combinational complexity [3].

The results show that it is now becoming viable to use exact synthesis as the optimization engine in the context of a larger logic synthesis algorithms. In fact, the algorithm described here has been used as a basis for various logic optimization algorithms [8], [12], [13]. Some of the algorithms build databases of optimal logic networks using exact synthesis, e.g.,

TABLE I: Exact synthesis of all 4-input NPN classes and a set of 6-input DSD functions. All runtimes are in milliseconds.

Function set	Nr. of functions	Mean runtime (ms)	Total runtime (ms)
NPN4	222	225.46	50052.12
FDSD6	1000	69.00	69000.00

all representatives of NPN classes, some of the algorithms find optimum network representations *on-the-fly*.

IV. Exact Delay Networks

This section discusses exact delay optimization using SAT-based exact synthesis. Exact delay synthesis can be achieved by extended the SAT formulation from the previous section. The extension allows for using a SAT solver to check whether there exists a Boolean network with r gates that realizes the n-variable functions $g_1, \ldots, g_m$ with a maximum delay of at most Δ assuming the input arrival times $\delta_1, \ldots, \delta_n$. Since logic rewriting is computing the delay and input arrival times based on the logic level, all values are integers.

A. Encoding

The idea is to assign a minimum depth to each gate and then constrain the maximum depth of the output gates. To encode the depth at each gate we make use of the *order encoding* [14], [5]. In the order encoding, a value x in the range $0 \leq x \leq M$ is represented by M variables x^l for $1 \leq l \leq M$ where $x^l = [x \geq l]$. We have $x = x^1 + x^2 + \cdots + x^M$, i.e., the bitstring derived by x^j has x ones followed by $(M - x)$ zeros. The clauses

$$(\bar{x}^{l+1} \vee x^l) \tag{2}$$

for $1 \leq l < M$ ensure this property.

Example 5: If $M = 4$, we can represent the values $x = 0, 1, 2, 3, 4$ by the bitstrings $0000, 1000, 1100, 1110, 1111$, respectively.

For the integration of delay constraints into the SAT formulation, we associate a value d_i with each gate to represent a lower bound on the delay of the gate x_i with $n < i \leq n + r$. The value is in the range $0 \leq d_i \leq \delta_{\max} + (i - n)$ where $\delta_{\max} = \max\{\delta_1, \ldots, \delta_n\}$ is the greatest input arrival time. Next, we can encode each d_i in the order encoding using variables d_i^l for $1 \leq l \leq \delta_{\max} + (i - n)$.

We add clauses to propagate the depth limits according to the wiring of the network. Clauses

$$\bigwedge_{l=1}^{\delta_{\max}+j-n} (\bar{s}_{ijk} \vee \bar{d}_j^l \vee d_i^{l+1}) \wedge \bigwedge_{l=1}^{\delta_{\max}+k-n} (\bar{s}_{ijk} \vee \bar{d}_k^l \vee d_i^{l+1}) \tag{3}$$

for $1 \leq j < k < i$ ensure that the minimum delay of gate x_i is larger by at least one compared to the minimum delay of the children, x_j and x_k. For $j \leq n$, the value of d_j^l is the constant value $[\delta_j \geq l]$. The same applies to values of d_k^l for $k \leq n$.

Finally, we add constraints $\bar{g}_{hi} \vee \bar{d}_i^{\Delta+1}$ for primary outputs. If a gate is a primary output, its depth must be less or equal to the maximum depth Δ. Note that this constraint only needs to be added if $\Delta < \delta_{\max} + (i - n)$; otherwise, the maximum depth constraint cannot be violated.

B. Enumeration-based method

As seen in the previous section, it is convenient to create a database of optimum solutions for logic optimization. Boolean function classification, such as NPN classification, can help to reduce the number of Boolean functions that need to be synthesized and stored. In exact delay synthesis the space of input instances is much large, as one needs to take into account also all possible delay values and input arrival time profiles. The work in [15] introduces the concept of equioptimizable input arrival time patterns which allow to compress the number of possible arrival time profiles and find databases that capture all possible delay configurations for all 4-input Boolean functions.

C. Experimental results

SAT-based exact delay synthesis has been used in a logic synthesis application to reduce the delay of large Boolean networks [16]. The algorithm enumerates all subnetworks with k inputs. For each subnetwork the input arrival time pattern and the current delay can be extracted from the network. Exact synthesis is then used to improve the current delay. If this is successful, the subnetwork is replaced by the optimized subnetwork. The approach has been applied for sizes with up to $k = 6$, where optimized networks are computed *on-the-fly*. For this size, a database solution is not feasible anymore [17], [15].

V. Gate Libraries and Structural Constraints

Besides optimizing for the traditional cost metrics such as size or depth of a Boolean logic network, exact synthesis can easily target optimization applications with many and complex constraints that need to be respected at the same time. Additional application constraints correspond to additional constraints added to the SAT formula. In fact, in a scenario in which the solution must ensure additional constraints, heuristic algorithms may be too weak. Heuristic algorithms can only find a solution that satisfies the constraints or that does not satisfy the constraint—it does not answer to the question whether a satisfying solution can exist at all. In this section, we present exact synthesis for problems with complex constraints and we address (i) constraints due to different primitives; and (ii) constraints due to the logic representations for which the synthesis has to be performed.

A. Encoding gate libraries

Previous sections considered exact synthesis using 2-input gate Boolean networks—meaning that each node of the logic network can represent *any* 2-input Boolean function. In this section we discuss constraints and clauses aiming at constraining the functionality of the nodes; further, we extend our notation to the 3-input case.

To constrain the functionality of the nodes, some clauses can be added to ensure only some Boolean functions to be used. For instance, for a 2-input Boolean network, only AND and OR gates, with possible input complementations, can be allowed with a further clause

$$(\bar{f}_{i11} \vee f_{i01} \vee f_{i10}) \tag{4}$$

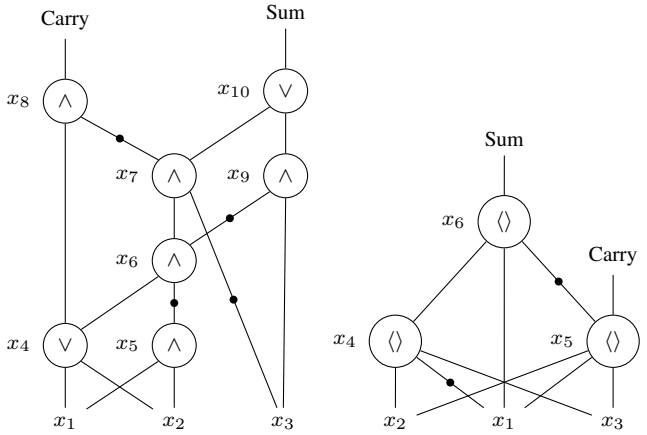

Fig. 2: (a) Boolean network for the full adder using only AND and OR operations; (b) Boolean network for the full adder using only 3-input majority operations and inversions. The $\langle\rangle$ is the majority operator, while bubbles on the edge represent complementation.

that excludes the f_{ipq} variables to realize the XOR gate. Recall the Boolean network from Fig. 1; the same full adder can be built using only AND and OR functions as shown in Fig. 2(a).

Next, we consider an extension of the SAT formulation to 3-input gate Boolean networks. In this case, the x_{it} and g_{hi} variables are used in the same way as in Section III-A, while both s_{ijk} and f_{ipq} need to be reexamined to work with three children. In the three input case, the new select variable is s_{ijkl}, which is true if the operands of gate x_i are x_j, x_k, and x_l. In a similar way, the function variables f_{ipqu} is true if the operation of gate x_i is true under the input assignment (p, q, u). Clauses can be built in a straightforward way starting from the ones depicted in Section III-B. It is worth noting that the variables f_{ipqu} allow for a representation of all 128 normal 3-input functions, but also in this case new clauses and variables can be added in order to limit the functionality of each node. Here, we address new variables and clauses in order to limit the functionality of each node to the 8 normal majority operation. As proposed in [13], a new variable operation can be added in order to encode which of these 8 operations is realized by each node. For each gate i, the operation variable o_{iw} is true if the operation of gate i is w, where w is one of the 8 possible normal majority operations. Two new clauses need to be added: (i) one to bind a function operation f_{ipqu} to the respective operation o_{iw}, and (ii) a clause that ensures that each gate realizes at least one of the 8 operations. Fig. 2(b) shows an optimum full adder, obtained by using only majority function and inversion (represented as complemented edges in the graph).

B. Encoding structural properties

Here, we consider the encoding of constraints due to structural properties. Examples of these constraints could be a limited depth (see Section IV), limited fan-out, or inversion patterns. Here, we propose a way to take into account fan-out limitations without changing the general encoding scheme of the problem; in other words, we describe further clauses that can added in order to limit the maximum fan-out of each node. One of the possible solutions that can be employed to constraint the maximum fan-out is by using a cardinality constraint to limit the maximum fan-out to a given value Φ. The fan-out of each node i can be encoded by limiting the number of select variables with index larger than i: $s_{(i+1)jkl}, \ldots, s_{(i+n)jkl}$ that are equal to 1. The constraint is forced on all select variables that use i as one of the children; moreover also the output variables need to be taken into account. As an example, consider Fig. 2(b). The cardinality constraint on Φ for node x_4 would be:

$$\begin{aligned} &s_{5124} + s_{5134} + s_{5234} + s_{6124} + s_{6134} + \\ &s_{6234} + s_{6145} + s_{6245} + s_{6345} + g_{14} + g_{24} \leq \Phi \end{aligned} \tag{5}$$

Given a concrete function, often it is helpful to include function-specific constraints into the SAT formula. Some functional decomposition properties may imply certain substructures or gate functions. Also arrival time profiles may prohibit or imply certain structures. Adding corresponding constraints to the SAT formula prunes the search space and can lead to significant performance improvements (see, e.g., [5], [16]).

C. Experimental results

We present experimental results using some of the constraints described above. The first two rows of Table II show experimental results for the synthesis of NPN4 and NPN5, using three-input Boolean networks (all the details can be found in [8]). The set NPN4 is the same used in previous sections, while the set NPN5 consists of 616,126 5-input NPN classes. The results show that minimum size 3-input Boolean networks can be found for all functions: 4-input functions can be synthesized in less than 0.5 seconds, while for NPN5 the average computation time is less than 10 minutes.

The last two rows of Table II show the exact synthesis for the two sets of functions used in Section III-E; in this case, we consider the following constraints to the 3-input Boolean network: (i) only majority functions can be employed; (ii) the maximum depth is limited to 3; (iii) the maximum fan-out is equal to 3. For the first set, a timeout of 10 minutes has been set; while a 2 minutes timeout is used for the 1000 functions. The "Nr. of timeouts" illustrates the number of functions that have not been finished in the given amount of time. All 222 NPN4 can be synthesized using a maximum depth of 3 and maximum fan-out of 3; none of the functions requires more than 10 minutes. For the FDSD6, 14 functions cannot be synthesized within the timeout.

VI. Related Work and Future Challenges

Exact synthesis has been studied thoroughly in the past. Ernst [18] provides a good overview of the related work in exact synthesis, which she organizes into three categories: i) algorithms based on functional decomposition (e.g., [19], [20], [21], [22]), ii) algorithms based on explicit (e.g, [23], [24], [25], [2]) or implicit network enumeration (e.g., [26], [5], [27], [28], [6], [7], and iii) hybrid approaches that combine both structural and functional properties (see, e.g., [18], [29], [30]). SAT-based exact synthesis as discussed in this paper is based on implicit enumeration; algorithms in this category

TABLE II: Exact synthesis of all 4-input NPN classes, 5-input NPN classes, and a set of 1000 6-input DSD functions using different constraints. All runtimes are reported in seconds.

Function set	Nr. of functions	Mean runtime (s)	Total runtime (s)	Nr. of timeout	Configuration
NPN4	222	0.001	0.432	0	3-input Boolean network
NPN5	616,126	553.294	5,506,943.478	0	3-input Boolean network
NPN4	222	1.391	305.960	0	3-input Boolean network, only majority, depth ≤ 3, fan-out ≤ 3
FDSD6	1000	1.269	1251.141	14	3-input Boolean network, only majority, depth ≤ 3, fan-out ≤ 3

are considered the most practical ones. One striking advantage of SAT-based implementations is that advances in new SAT solvers directly influence the performance of exact synthesis. Also, SAT formulas versatile and easily allow the integration of additional constraints.

Being based on SAT provides ample freedom on how to encode the synthesis problem into a SAT formula—a choice that has significant impact on the performance. Finding an optimal encoding, in general or with respect to a given problem instance, is unsolved and has barely been addressed so far. Further, the runtimes of individual instances can highly differ, even when the problem sizes are similar: while some problems may take a few micro-seconds to solve, others may take several hours. Such irregularity impedes the integration into robust logic synthesis flows. While initial solutions to address this problem have been presented [12], [8], finding more solutions to handle or partition difficult instances remains of high interest.

Several open source implementations for SAT-based exact synthesis are available online. ABC [31] has four commands '`exact`', '`twoexact`', '`lutexact`', and '`majexact`'. A variant with a different encoding for the structural constraints is available at github.com/whaaswijk/topsynth. Further, in CirKit (github.com/msoeken/cirkit), there are SMT-based exact synthesis algorithms '`exact_mig`' and '`exact_xmg`' to find optimum majority-inveter graphs and optimum XOR-majority graphs, respectively.

REFERENCES

[1] T. Sasao, *Switching Theory for Logic Synthesis*. Springer, 1999.

[2] D. E. Knuth, *The Art of Computer Programming, Volume 4A*. Addison-Wesley, 2011.

[3] C. E. Shannon, "The synthesis of two-terminal switching circuits," *Bell System Technical Journal*, vol. 28, no. 1, pp. 59–98, 1949.

[4] A. Biere, M. Heule, H. van Maaren, and T. Walsh, Eds., *Handbook of Satisfiability*. IOS Press, 2009.

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

[6] A. Kojevnikov, A. S. Kulikov, and G. Yaroslavtsev, "Finding efficient circuits using SAT-solvers," in *Int'l Conf. on Theory and Applications of Satisfiability Testing*, 2009, pp. 32–44.

[7] N. Éen, "Practical SAT - a tutorial on applied satisfiability solving," 2007, slides of invited talk at FMCAD.

[8] W. Haaswijk, M. Soeken, L. G. Amarù, P.-E. Gaillardon, and G. De Micheli, "A novel basis for logic optimization," in *Asia and South Pacific Design Automation Conference*, 2017, pp. 151–156.

[9] W. Haaswijk, E. Testa, M. Soeken, and G. De Micheli, "Classifying functions with exact synthesis," in *Int'l Symp. on Multiple-Valued Logic*, 2017, pp. 272–277.

[10] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," in *Computer Aided Verification*, 2000, pp. 154–169.

[11] V. Bertacco and M. Damiani, "Boolean function representation based on disjoint-support decompositions," in *Int'l Conf. on Computer Design*, 1996, pp. 27–32.

[12] M. Soeken, L. G. Amarù, P.-E. Gaillardon, and G. De Micheli, "Exact synthesis of majority-inverter graphs and its applications," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 36, no. 11, pp. 1842–1855, 2017.

[13] E. Testa, M. Soeken, O. Zografos, F. Catthoor, and G. De Micheli, "Exact synthesis for logic synthesis applications with complex constraints," in *Int'l Workshop on Logic and Synthesis*, 2017.

[14] J. M. Crawford and A. B. Baker, "Experimental results on the application of satisfiability algorithms to scheduling problems," in *National Conf. on Artificial Intelligence*, 1994, pp. 1092–1097.

[15] L. G. Amarù, M. Soeken, P. Vuillod, J. Luo, A. Mishchenko, P.-E. Gaillardon, J. Olson, R. K. Brayton, and G. De Micheli, "Enabling exact delay synthesis," in *Int'l Conf. on Computer-Aided Design*, 2017.

[16] M. Soeken, G. De Micheli, and A. Mishchenko, "Busy man's synthesis: Combinational delay optimization with SAT," in *Design, Automation and Test in Europe*, 2017, pp. 830–835.

[17] W. Yang, L. Wang, and A. Mishchenko, "Lazy man's logic synthesis," in *Int'l Conf. on Computer-Aided Design*, 2012, pp. 597–604.

[18] E. A. Ernst, "Optimal combinational multi-level logic synthesis," Ph.D. dissertation, The University of Michigan, 2009.

[19] R. M. Karp, F. E. McFarlin, J. P. Roth, and J. R. Wilts, "A computer program for the synthesis of combinational switching circuits," in *Symp. on Switching Circuit Theory and Logical Design*, 1961, pp. 182–194.

[20] J. P. Roth and R. M. Karp, "Minimization over Boolean graphs," *IBM Journal of Research and Development*, vol. 6, no. 2, pp. 227–238, 1962.

[21] P. R. Schneider and D. L. Dietmeyer, "An algorithm for synthesis of multiple-output combinational logic," *IEEE Trans. on Computers*, vol. 17, no. 2, pp. 117–128, 1968.

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

[23] L. Hellerman, "A catalog of three-variable Or-invert and And-invert logical circuits," *IEEE Trans. Electronic Computers*, vol. 12, no. 3, pp. 198–223, 1963.

[24] R. A. Smith, "Minimal three-variable NOR and NAND logic circuits," *IEEE Trans. Electronic Computers*, vol. 14, no. 1, pp. 79–81, 1965.

[25] R. Drechsler and W. Günther, "Exact circuit synthesis," in *Int'l Workshop on Logic and Synthesis*, 1998.

[26] S. Muroga and T. Ibaraki, "Design of optimal switching networks by integer programming," *IEEE Trans. on Computers*, vol. 21, no. 6, pp. 573–582, 1972.

[27] C. R. Baugh, C. S. Chandersekaran, R. S. Swee, and S. Muroga, "Optimal networks of NOR-OR gates for functions of three variables," *IEEE Trans. on Computers*, vol. 21, no. 2, pp. 153–160, 1972.

[28] S. Muroga and H. C. Lai, "Minimization of logic networks under a generalized cost function," *IEEE Trans. on Computers*, vol. 25, no. 9, pp. 893–907, 1976.

[29] E. S. Davidson, "An algorithm for NAND decomposition under network constraints," *IEEE Trans. on Computers*, vol. 18, no. 12, pp. 1098–1109, 1969.

[30] 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. on Computers*, vol. 28, no. 1, pp. 76–85, 1979.

[31] R. K. Brayton and A. Mishchenko, "ABC: an academic industrial-strength verification tool," in *Computer Aided Verification*, 2010, pp. 24–40.