SAT Based Exact Synthesis using DAG Topology Families

Winston Haaswijk
EPFL
Lausanne, Vaud, Switzerland
winston.haaswijk@epfl.ch

Alan Mishchenko
University of California, Berkeley
Berkeley, California, United States
alanmi@berkeley.edu

Mathias Soeken
EPFL
Lausanne, Vaud, Switzerland
mathias.soeken@epfl.ch

Giovanni De Micheli
EPFL
Lausanne, Vaud, Switzerland
giovanni.demicheli@epfl.ch

ABSTRACT

SAT based exact synthesis is a powerful technique, with applications in logic optimization, technology mapping, and synthesis for emerging technologies. However, its runtime behavior can be unpredictable and slow. In this paper, we propose to add a new type of constraint based on families of DAG topologies. Such families restrict the search space considerably and let us partition the synthesis problem in a natural way. Our approach shows significant reductions in runtime as compared to state-of-the-art implementations, by up to 63.43%. Moreover, our implementation has significantly fewer timeouts compared to baseline and reference implementations, and reduces this number by up to 61%. In fact, our topology based implementation dominates the others with respect to the number of solved instances: given a runtime bound, it solves at least as many instances as any other implementation.

ACM Reference Format:
Winston Haaswijk, Alan Mishchenko, Mathias Soeken, and Giovanni De Micheli. 2018. SAT Based Exact Synthesis using DAG Topology Families. In *DAC '18: DAC '18: The 55th Annual Design Automation Conference 2018, June 24–29, 2018, San Francisco, CA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3195970.3196111

1 INTRODUCTION

In recent years, there has been a large research effort into SAT based logic synthesis methods. A notable research direction has been the synthesis of optimum Boolean chains, also known as *exact synthesis* [4, 16, 17]. A Boolean chain is a *directed acyclic graph* (DAG), in which every vertex corresponds to a 2-input Boolean operator [10]. Boolean chains are compact structures for the representation of multiple-output Boolean functions. They are similar to the concept of *unbound logic networks* used by the logic synthesis community. Exact synthesis has various applications in logic optimization, technology mapping, and synthesis for emerging technologies [4, 14, 16, 17]

Given a Boolean function, we can solve the problem of finding an optimum Boolean chain for that function by solving sequences of SAT formulae [3, 12]. However, the major drawback of SAT based synthesis is the unpredictable, and potentially long, runtime of SAT invocations. This runtime is perhaps unsurprising if we consider that, in finding optimum Boolean chains, the SAT solver has to simultaneously perform at least two distinct tasks:

(1) finding valid DAG structures for the Boolean chain
(2) assigning Boolean operators to the vertices in these DAGs, such that the entire sequence of the chain corresponds to the specified Boolean function

Another drawback to SAT based synthesis is that, like many logic synthesis and EDA problems, it is difficult to parallelize. Some efforts have been made in parallelizing SAT solvers using techniques such as *cube-and-conquer*, clause sharing, and so-called *portfolio* SAT solvers that apply different SAT solvers in a parallel or distributed manner [6, 7]. However, the search space is so large that the impact of using multiple threads is limited. Moreover, parallel SAT solvers based on these methods are largely domain agnostic, and do not take advantage of the structure of specific problem domains.

The main contribution of this paper is the proposal to use DAG topology families to mitigate the runtime problems of exact synthesis. These topology families can be used to provide additional constraints to the SAT solver, and thus to speed up the synthesis process. In other words, topology based synthesis is a proposal to synthesize Boolean chains (and extensions thereof) while providing additional aid to the SAT solver in performing step (1).

The rest of this paper is organized as follows. Section 2 provides some background on Boolean chains, and on finding optimum Boolean chains using SAT based exact synthesis. In Section 3 we introduce a simple generalization of Boolean chains which allows for chains that have n-input operators. Then, in Section 4, we discuss DAG topology families and introduce the related concept of Boolean *fences*. We discuss some theoretical properties of fences and algorithms for generating them. Next, in Section 5, we show how the conventional SAT based exact synthesis algorithm can be extended to include topological constraints. Then, in Section 6 we perform several experiments with our new synthesis algorithm and compare it to the state-of-the-art. We show that our topology based algorithm is able to obtain significant runtime reductions of up to 63.43%. Perhaps even more importantly, given the runtime budget, it also is able to solve up to 61% more problem instances. Moreover, our topology based algorithm dominates the others in terms of solved instances. Finally, we conclude the paper with a brief discussion in Section 7.

DAC '18, June 24–29, 2018, San Francisco, CA, USA

ACM ISBN 978-1-4503-5700-5/18/06…$15.00
https://doi.org/10.1145/3195970.3196111

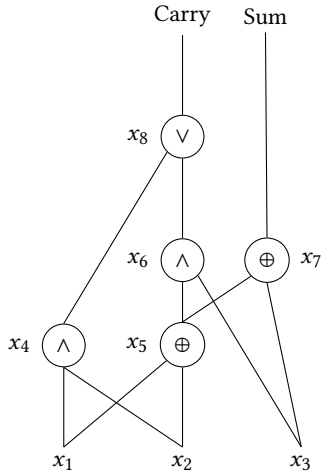

Figure 1: Illustration of a Boolean chain for a full adder. As it is not used, the constant zero input x_0 is not shown here.

2 BACKGROUND

2.1 Boolean Chains

Suppose we are given m functions $f_1(x_1, \ldots, x_n), \ldots, f_m(x_1, \ldots, x_n)$ that each compute some values for the same set of n inputs. As described by Knuth in [10], a Boolean chain for these functions is a sequence $(x_{n+1}, \ldots, x_{n+r})$ with the property that each step in the sequence combines two of the preceding steps:

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

where $1 \leq j(i) < i$, and $1 \leq k(i) < i$, and where $\circ_i$ is one of sixteen binary operators. Such a chain must also have the property that for $1 \leq j \leq m$, either $f_j(x_1, \ldots, x_n) = x_{l(j)}$ or $f_j(x_1, \ldots, x_n) = \bar{x}_{l(j)}$, where $0 \leq l(j) \leq n+r$ and $x_0 = 0$. Since each step can only refer to previous steps in the sequence, there is a partial order between steps and we can view such chains as DAGs.

For example, when $n = 3$, then the chain

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

can be used to represent the 3-input 2-output function $f(x_1, x_2, x_3) = (x_1 \oplus x_2 \oplus x_3, \langle x_1, x_2, x_3 \rangle)$, which is commonly known as a full adder. Figure 1 illustrates this example.

2.2 SAT Based Exact Synthesis

We present here a variant of Knuth's algorithm for the synthesis for optimum Boolean chains [11]. There are different variations on this algorithm, for example with different *selection variable* schemes [12]. In all variations, however, the key idea behind SAT formulation is the same: to construct a SAT formula that is satisfiable *if and only if* there exists a Boolean chain with r steps that computes functions $f_1, \ldots, f_m$ depending on n variables. In Knuth's algorithm, such a formula consists of the following variables, for $1 \leq h \leq m$, $n < i \leq n+r$, and $1 \leq t \leq 2^n$:

$$x_{it} : \text{t}^{\text{th}} \text{ bit of } x_i\text{'s truth table}$$
$$g_{hi} : f_h(x_1, \ldots, x_n) = x_i$$
$$s_{ijk} : x_i = x_j \circ_i x_k \text{ for } 1 \leq j < k < i$$
$$f_{ipq} : p \circ_i q \text{ for } 0 \leq p, q \leq 1$$

Here, the g_{hi} variables determine which outputs point to which steps. The s_{ijk} variables determine the inputs j and k, for each step i. These are also known as *selection variables*. The f_{ipq} encode for all steps i what the corresponding Boolean operator is.

These variables are then constrained by a set of clauses which ensure that the chain computes the correct functions. For $0 \leq a, b, c \leq 1$ and $1 \leq j < k < i$, the main clauses are:

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

Intuitively, these clauses encode the following constraint: if step i has inputs j and k, *and* the t^{th} bit of x_i is a, *and* the t^{th} bit of x_j is b, and the t^{th} bit of x_k is c, *then* it must be the case that $b \circ_i c = a$. This can be understood by rewriting the formula as follows:

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

Note that a, b, and c are constants which are used to set the proper variable polarities.

Next, let $(t_1, \ldots, t_n)_2 = t$ be the binary encoding 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 $f_h(t_1, \ldots, t_n)$. We also add $\bigvee_{i=n+1}^{n+r} g_{hi}$ and $\bigvee_{k=1}^{i-1} \bigvee_{j=1}^{k-1} s_{ijk}$, so that every output points to a step in the chain and to ensure that every step has two inputs.

These are the only clauses necessary to ensure that a valid chain is found. Other clauses, such as forcing a colexicographic order on the steps, may be added to speed up synthesis. We refer the interested reader to [11].

One can use these clause to synthesize Boolean chain as follows:[1]

(1) Initialize $r = 1$.
(2) Generate the clauses for an r step chain.
(3) If they are satisfiable, we are done.
(4) Otherwise, increment r and goto 2.

The final value for r represents the minimum number of steps necessary to compute the specified functions.

3 GENERALIZED BOOLEAN CHAINS

In Section 2.1 we describe the common definition of Boolean chains. However, that definition includes only binary operators. We can extend the definition to include Boolean operators of arbitrary numbers of variables. For simplicitly, we will impose the restriction that the number of operands of the Boolean operators will be upper bounded by some fixed n.

This extension has two main motivations. First, synthesis of chains with larger operators may be significantly faster. For example, using Boolean chains with 3-input operators, one can efficiently classify the set of all 5-input functions using SAT based exact synthesis [5], whereas this has not been achieved for 2-input operator chains. Second, one application of exact synthesis is in technology mapping, where we are often required to use a diverse set of logic *primitives*. For example, we may be given a standard cell library and asked to find an efficient implementation of a function f in

[1] For convenience, we do not consider trivial special cases such as 0-step chains here.

terms of the primitive Boolean operators defined by the standard cells. Generally, we cannot assume that the cell library contains only 2-input operators. Following the convention of [15], we denote such a set of primitives by $\mathcal{B}$.

We extend the definition as follows. Let $f = (f_1, \ldots f_m)$ be a multiple-output Boolean function, such that $f : \mathbb{B}^n \rightarrow \mathbb{B}^m$. An *unbounded* Boolean chain is a sequence $(x_{n+1}, \ldots, x_{n+r})$:

$$x_i = \phi_i(x_{j(i,1)}, \ldots, x_{j(i,\iota(\phi_i))}) \qquad \text{for } n+1 \leq i \leq n+r$$

such that $1 \leq j(i,v) < i$, and $1 \leq \iota(\phi_i) \leq n$, and for all $1 \leq k \leq m$, either $f_k(x_1, \ldots, x_n) = x_{l(k)}$ or $f_k(x_1, \ldots, x_n) = \bar{x}_{l(k)}$, where $0 \leq l(k) \leq n+r$, and $x_0 = 0$. We call $\iota(\phi_i)$ the *fanin* of operator ϕ_i. For convenience, in the following we use Boolean chains to refer to this extended definition. Note that for all Boolean chains $n < i \leq n+r \Rightarrow \phi_i \in \mathcal{B}$. For example, in the common definition of Boolean chains, we have $\iota(\phi_i) = 2$ for all i and $\mathcal{B}$ is the set of all binary operators.

4 DAG TOPOLOGIES, FENCES, AND FAMILIES

Suppose we are given a DAG $G = (V, E)$, and a Boolean function $f : \mathbb{B}^n \rightarrow \mathbb{B}^m$. We may be able to transform the DAG into a Boolean chain for f by assigning the appropriate operators $\phi_i \in \mathcal{B}$ to every vertex $v_i \in V$. We call such a transformation a *labeling* of the graph. Recall that finding a labeling corresponds to step (2) of the synthesis process described in the introduction. Finding such a labeling may not be possible, but if it exists, a SAT solver can find it efficiently. For example, consider the 6-input function with truth table 0x9ef7a8d9c7193a0f. The smallest known implementation of this function uses 19 2-input gates. When the solution topology is given, a SAT solver can find a labeling in 0.12s on a laptop computer.

The efficiency of labeling may inspire one to think of the following (naive) synthesis algorithm:

```
function Synthesize(f)
    while true do
        G ← NextDag()
        if LabelingExists(G, f) then
            Chain ← LabelGraph(G, f)
            return Chain
        end if
    end while
end function
```

Such an algorithm reduces to efficiently finding a DAG with the proper structure for f. However, in general, given f we do not know a priori which DAG structures have a labeling for f. Given an n-input function, finding a suitable DAG requires us to search a very large space of DAG structures. Unfortunately, the enumeration of potential DAGs in this space generally outweighs the potential efficiency of graph labeling.

Alternatively, instead of providing the SAT solver with a DAG to label, we can specify a set of clauses which constrain the SAT solver's search to a particular family of DAG topologies. We then use the SAT solver's efficient search heuristics to find only those topologies within that family. This approach avoids explicit enumeration of DAGs and provides a middle ground between the unstructured exact synthesis formulation of Section 2.2 and the fully structured labeling of graphs. The idea to explore this middle ground is the main contribution of this paper.

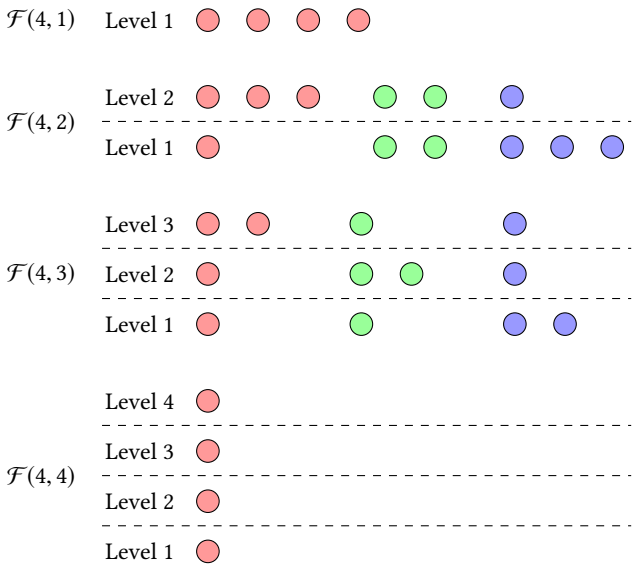

Figure 2: An illustration of the fences in $\mathcal{F}_4$. Every fence corresponds to a family of DAGs with the same distribution of nodes accross levels. We draw an edge here between the first node on each level in a fence, just for visualization purposes.

4.1 Fences

Given two integers k and l ($1 \leq l \leq k$), a *Boolean fence* is a partition of k nodes over l levels, where every level contains at least one node. We can denote a Boolean fence by an ordered sequence $F = (\lambda_1, \ldots, \lambda_l)$, where every λ_i corresponds to the collection of nodes on level i. A Boolean fence (k, l) is not unique: there may be multiple ways of distributing k nodes over l levels. We call the set of all such partitions a Boolean fence *family* and write $\mathcal{F}(k, l)$. We use $\mathcal{F}_k$ to denote the set of all fence families of k nodes:

$$\mathcal{F}_k = \{\mathcal{F}(k,l) \mid 1 \leq l \leq k\}$$

To be concise, we also refer to Boolean fences and fence families as fences and families, respectively. Boolean fences can be visualized as graphs. Figure 2 shows the fences in $\mathcal{F}_4$.

Every DAG of n nodes corresponds to a unique fence $F \in \mathcal{F}_n$. To see why, note that we can assign levels to nodes in a DAG based on their partial order. Such an assignment allows us to find the level distribution corresponding to the fence F.

A fence induces a set of DAG topologies, in which each topology corresponds to the same distribution of nodes over levels, but with different arcs between nodes. In other words, fences are representations for families of graph topologies. Consequently, a fence also induces a set of Boolean chains with those topologies.

4.2 Counting Fences

Let us consider the following question: how many fences are there in family $\mathcal{F}(k, l)$? Note that, in this family, l nodes are fixed, since we need to have at least one node on l levels. The remaining $k-l$ nodes may be arbitrarily distributed across the l levels. In other words, our question reduces to: how many ways are there to distribute $k-l$ indistinguishable nodes across l bins? The answer is equal to the

number of nonnegative integer-valued solutions to the equation

$$x_1 + x_2 + \cdots + x_l = k - l$$

and hence

$$|\mathcal{F}(k,l)| = \binom{k-1}{l-1}. \quad (1)$$

We can now use Formula 1 to count the total number of fences of k nodes, $|\mathcal{F}_k|$ as follows:

$$|\mathcal{F}_k| = \sum_{i=1}^{k} \binom{k-1}{i-1} = 2^{k-1}$$

The reader may verify that these formulas correctly predicts the numbers of fences in Figure 2.

4.3 Generating Fences

Our synthesis method requires an efficient algorithm for the generation of fences. Suppose we want to generate $\mathcal{F}_k$. In order to do this, we first observe that the number of fence families in $\mathcal{F}_k$ closely correspond to different integer partitionings of k. Recall that, given an integer k, an integer partition of k is a way of writing k as the sum of positive integers $k_1 + \cdots + k_i = k$. We can obtain a fence from such a partition by imposing an order on it. Let S be the multiset of integers corresponding to an integer partition of k, and let $l = |S|$. Now, we can create a fence $F \in F(k,l)$ from this partition by fixing $F = (k_1, \ldots, k_l)$ where $k_i \in S (1 \leq i \leq l)$. Note that S is a unique partition of k. However, F may not be the only fence corresponding to this partition. To see why, let π be a permutation of l. Then, the fence $F' = F_\pi = (k_{\pi(1)}, \ldots, k_{\pi(n)})$ is also a fence in $F(k,l)$.

Thus, to generate all fences in $\mathcal{F}_k$, we have to do the following:

- Generate all integer partitions S of k.
- For all such S, generate all permutations π_S.

In practice, we are often not interested in enumerating all 2^{k-1} fences in $\mathcal{F}_k$. Instead, we are often satisfied once we obtain a fence that our synthesizer finds a solution with. All of this suggest the lazy fence generating algorithm in Algorithm 1. The algorithm presented here is a coroutine that may be called repeatedly and yields all fences in $\mathcal{F}_k$ until exhausted. The algorithm is constructed by composing standard integer partitioning and permutation algorithms. In our implementation we use a lazy adaptation of the integer partition algorithm from Knuth [10] (fascicle 3, page 38), who attributes it to Hindenburg [8]. For the permutations we use an algorithm from the C++ standard library.

Algorithm 1 is an efficient procedure that may be function in the inner loop of a synthesis algorithm. For example, we can generate set $\{\mathcal{F}_k \mid k \leq 10\}$ in 0.097 seconds. On top of this basic procedure we can also build more sophisticated algorithms, such as algorithms that filter out any fences that are unnecessary for a specific synthesis task. We discuss such methods in Section 5.

Algorithm 1 An algorithm to generate all fences in $\mathcal{F}_k$.

```
function GenerateFences(k)
    while true do
        S ← NextPartition(k)
        if S ≠ ∅ then
            l ← |S|
            while true do
                π ← NextPermutation(S)
                if π ≠ ∅ then
                    F ← EmptyFence(l)
                    for i ← 0; i < l; i++ do
                        F[i] ← S[π(i)]
                    end for
                    yield F
                else
                    break
                end if
            end while
        else
            yield ∅
        end if
    end while
end function
```

5 EXACT SYNTHESIS USING FENCES

We have seen how fences correspond to families of DAG topologies, investigated some of their theoretical properties, and presented a fence generating algorithm. In this section we consider how to use fences to accelerate exact synthesis by using them to provide additional constraints in the SAT formulation. To do so, let us first look at some connections between fences and Boolean chains.

Consider a fence $F = (\lambda_1, \ldots, \lambda_l)$. Let $G = (V, E)$ be a DAG, and let $\tau(v) : V \to \mathbb{N}$ be the function that assigns each vertex from G to its level. Let $\tau_i = |\{v \mid \tau(v) = i\}|$. We say that G *satisfies* F if and only if $|\lambda_i| = \tau_i$. In other words, a DAG satisfies the topological constraints of a fence if its distribution of nodes across levels is the same. We say that a Boolean chain satisfies F if its underlying DAG structure satisfies F. We consider the primary inputs of the chain to have level 0, and do not consider them in satisfying F.

For example, consider the fence $F = (\lambda_1, \lambda_2) \in F(4,2)$ highlighted in Figure 3(a). We have numbered its nodes to make them easier to distinguish. Intuitively, only DAGs with two nodes on the first level and two nodes on the second level satisfy F. For example, Figure 3(b) is a 2-input operator Boolean chain satisfying the constraints from F. Similarly Figure 3(c) is a 3-input Boolean chain that satisfies F. However, Figure 3(d) shows a chain that is invalid for F. It violates the constraint that the step corresponding to fence node 4 be on level 2.

Observe that the topology constraints captured by fences are independent of number of inputs, or operator fanin. This is desirable, as it implies that the same fence generator can be used as the basis for synthesis of generalized Boolean chains and functions of arbitrary input size.

Now consider again the arbitrary fence $F = (\lambda_1, \ldots, \lambda_l) \in F(k,l)$. Suppose we wish to synthesize a Boolean chain that satisfies F. We know that it must be a k-step chain. We assign step i to level t by setting

$$\tau(x_i) = t \Leftrightarrow t = \min_{t'} i \leq \sum_{j=0}^{t'} |\lambda_j|.$$

where $|\lambda_0| = n$, the number of primary inputs.

Note that if $\tau(x_i) = t$, then step x_i must, by definition, have at least one fanin on level $t-1$. Thus, the fence constrains not only

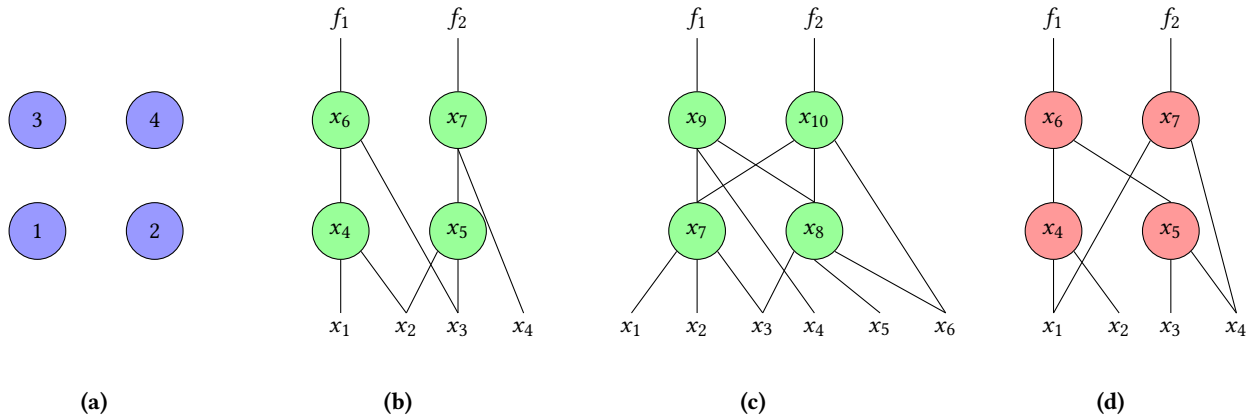

Figure 3: The fence F in (a) corresponds to a set of possible DAG topologies and can thus be used to constrain the SAT solver's search. For instance, Figure (b) and Figure (c) satisfy the constraints from F. Figure (d) does not.

the distribution of nodes accross levels, but also the fanin relations between nodes. Due to this level constraint, in the SAT formulation the selection variable s_{ijk} may never be true if $\tau(k) < t - 1$, for any $i < k$. Let k' and k'' be the smallest and largest indices such that $\tau(x_{k'}) = t - 1$ and $\tau(x_{k''}) = t - 1$, respectively. A simple way to express the constraints imposed by the fence is by adding, for each step x_i, the clause $\bigvee_{k=k'}^{k''} s_{ijk}(j < k)$. In that way, we ensure that each step has at least one fanin from a level directly below. This approach is similar to the way that colexicographic or other symmetry-breaking clauses are added in [11]. However, we can do better. Observe that, since none of the variables outside of $\{s_{ijk} \mid k' \leq k \leq k''\}$ may be true, we do not need to include them in our SAT formula at all. Thus, by using a fence we can significantly reduce the number of variables and clauses in our SAT instances.

To implement exact synthesis with topological constraints we can then proceed as follows: **(i)** Generate a new fence using the procedure described in Algorithm 1. **(ii)** Using the constraints implied by the fence, generate a reduced SAT formula. We use a set of clauses analogous to the one described in Section 2.2. However, we exclude any variables or clauses that are rendered unnecessary due to the fence constraints, obtaining a simpler SAT formula. **(iii)** If the formula is satisfiable, we are done. **(iv)** Otherwise, go to **(i)**.

The fence generation procedure in Algorithm 1 can be easily modified to yield a stream of fences of increasing size. Thus, we extend the conventional exact synthesis algorithm, while decomposing the search space using increasingly large families of graph topologies. Recall that in Section 4.2 we derived the total number of fences of k nodes. Given an upper bound on the number of nodes to realize a function, we therefore also have an upper bound on the number of decomposed exact synthesis instances we have to solve.

6 EXPERIMENTS

In order to evaluate the performance of our proposed approach, we find exact synthesis for the following collection of Boolean function sets:

- *NPN4:* All 222 4-input NPN classes [9].
- *FDSD6:* 1000 fully-DSD decomposable 6-input functions. that occur frequently in practical synthesis and technology mapping applications [13].
- *PDSD6:* 1000 *partially*-DSD decomposable 6-input functions that occur frequently in practical technology mapping applications.
- *FDSD8:* 100 fully-DSD decomposable 8-input functions.
- *PDSD8:* 100 partially-DSD decomposable 8-input.

We compare five different SAT based exact synthesis approaches, including a state-of-the-art reference implementation from the well known ABC logic synthesis package [1].

(1) *SYM:* A baseline implementation of the SAT-based exact synthesis algorithm, which is very similar in architecture and performance to ABC's state-of-the-art implementation (command `exact`, [17]).
(2) *SYM-CEGAR:* Approach SYM with CEGAR loop to add the main clauses incrementally [2].
(3) *TOP:* Our proposed algorithm based on fence enumeration and the use of additional topological constraints.
(4) *TOP-CEGAR:* Approach TOP with CEGAR loop to add the main clauses incrementally.
(5) *ABC:* The state-of-the-art reference implementation in ABC.

Table 1 lists all experimental results. For each approach four values are listed: i) the mean solving time (*mean*), ii) the standard deviation (*dev*), both in milliseconds, iii) the number of instances that could not be solved before timing out (*#t/o*), and the number of instances that were successfully solved (*#ok*). Note that the number of solved instances is the most important metric here, as it captures in essence how practical an algorithm is. Given a bound on runtime, we obviously prefer the algorithm that can solve the most problems within that bound. A similar metric is commonly used in SAT solver competitions. In our experiments we limit runtime by specifying a conflict limit to each SAT instance. All experiments were performed on an Intel Xeon E5-2680 v3 (Haswell) 2.5 GHz processor with a 30 MB cache and 256GB of RAM.

The results in Table 1 show that using topological structure enumeration can significantly improve the solving time, as well as the number of solved instances. For *NPN4*, our topology based

Table 1: Experimental Results (all runtimes in milliseconds)

Functions	SYM				SYM-CEGAR				TOP				TOP-CEGAR				ABC			
	mean	dev	#t/o	#ok	mean	dev	#t/o	#ok	mean	dev	#t/o	#ok	mean	dev	#t/o	#ok	mean	dev	#t/o	#ok
NPN4	266.48	717.68	0	222	225.46	696.40	0	222	216.70	480.16	0	222	216.69	420.53	0	222	**177.69**	581.67	0	222
FDSD6	96.55	69.21	0	1000	69.00	57.43	0	1000	39.07	18.93	0	1000	**29.61**	**16.19**	0	1000	77.16	63.88	0	1000
PDSD6	43453.33	64757.49	256	744	43453.33	64757.49	1000	0	**20707.11**	**32590.64**	**128**	**872**	21605.81	33243.84	256	744	54525.05	85070.54	256	744
FDSD8	11998.39	7921.64	0	100	5583.13	3309.73	0	100	4809.29	2714.85	0	100	2688.51	**1383.34**	0	100	**2490.18**	1825.72	0	100
PDSD8	214959.90	86964.25	14	86	150533.31	75998.48	42	58	100871.79	62671.56	**11**	**89**	**78619.48**	**32737.19**	40	60	137010.13	41097.86	76	24

algorithm is more than **19%** faster than our baseline implementation. All algorithms find the solutions for all problem instances. For *FDSD6*, *TOP-CEGAR* is **57.09%** faster than *SYM-CEGAR* and **61.63%** faster than *ABC*. Again, there are no timeouts. For *PDSD6*, *TOP* is **52.35%** faster than *TOP* and **62.02%** faster than *ABC*. We can also solve **50%** more instances than both other approaches before timing out. The same observation can be made for the 8-input function sets. For *FDSD8*, *TOP-CEGAR* is up to **51.85%** faster than *SYM-CEGAR*. For *PDSD8*, *TOP-CEGAR* is **63.43%** faster than *SYM-CEGAR* and **42.68%** faster than *ABC*. Again, our algorithm has fewer timeouts than the others. Note especially the striking difference in timeout percentage with the ABC implementation. Our algorithm has **61%** fewer timeouts than ABC. In fact, the table shows that it dominates the other implementations with respect to the number of solved instances.

In summary, we can see that the gains from using topological constraints can be quite significant. They seem to be particularly beneficial as the functions to be synthesized become larger and harder to synthesize.

7 DISCUSSION & FUTURE WORK

This paper takes a new look at the difficult problem of SAT-based exact synthesis for Boolean functions. The SAT-based exact synthesis formulation must encode both the structure of the logic network and the functionality of its nodes. These are cleverly encoded using SAT clauses, however, their combination makes the problem very hard to solve. We find that knowing the structure makes the problem significantly easier. Based on this observation, we introduce a SAT-based exact synthesis method based on topological structure enumeration. Since the number of topological structures grows very quickly as the number of gates increases, we collect a set of structures in what we call a Boolean fence. The paper introduces a theory of Boolean fences, illustrates how they are enumerated, and shows how they are used to constrain the SAT-based exact synthesis encoding.

We evaluate our new approach to find optimum logic networks for various sets of practical Boolean functions. Our experimental results show a runtime reduction of up to **63.43%**. Further, we find find many solutions for instances in which the state-of-the-art method, which does not make use of topological constraints, times out. We reduce the number of timeouts by up to **61%**. Our topology based algorithm *dominates* the other implementations in the number of solved instances: given a runtime budget it always solves at least as many as any other implementation. This has direct impact on a variety of logic optimization algorithms that use exact synthesis, such as logic rewriting, technology mapping, and synthesis for emerging technologies[4, 14, 16, 17].

In future work, we plan to find enumerate topological structures according to different strategies to reach further speed-ups. Furthermore, DAG topologies provide a natural way of partitioning the synthesis search space: only parts of the space containing potentially valid topologies needs to be examined. An interesting line of investigation would be to use graph topologies as the basis for parallel SAT based exact synthesis algorithms, using the graph topologies as a source of parallelism.

ACKNOWLEDGMENTS

This work was supported in part by SRC contract 2710.001 "SAT-based methods for scalable synthesis and verification" at UC Berkeley, H2020-ERC-2014-ADG 669354 CyberCare, and the Swiss National Science Foundation (200021-169084 MAJesty).

REFERENCES

[1] Robert K. Brayton and Alan Mishchenko. 2010. ABC: An Academic Industrial-Strength Verification Tool. In *Computer Aided Verification*. 24–40.
[2] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. 2000. *Counterexample-Guided Abstraction Refinement*. Springer Berlin Heidelberg, Berlin, Heidelberg, 154–169. https://doi.org/10.1007/10722167_15
[3] Niklas Eén. 2007. Practical SAT - a tutorial on applied satisfiability solving. In *FMCAD*.
[4] Winston Haaswijk, Mathias Soeken, Luca Amarú, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli. 2017. A Novel Basis for Logic Rewriting. In *ASPDAC*.
[5] Winston Haaswijk, Eleonora Testa, Mathias Soeken, and Giovanni De Micheli. 2017. Classifying Functions with Exact Synthesis. In *ISMVL*.
[6] Youssef Hamadi. 2009. ManySAT : a Parallel SAT Solver. *Journal on Satisfiability, Boolean Modeling and Computation* 6, 5 (2009), 245–262. https://doi.org/10.1152/japplphysiol.00460.2010
[7] Marijn J. H. Heule, Oliver Kullmann, Siert Wieringa, and Armin Biere. 2012. *Cube and Conquer: Guiding CDCL SAT Solvers by Lookaheads*. Springer Berlin Heidelberg, Berlin, Heidelberg. 50–65 pages. https://doi.org/10.1007/978-3-642-34188-5_8
[8] C.E. Hindenburg. 1779. *In Nitinomii Dignitatum Exponentis Indeterminati*. Ph.D. Dissertation. University of Göttingen.
[9] Zheng Huang, Lingli Wang, Yakov Nasikovskiy, and Alan Mishchenko. 2013. Fast Boolean matching based on NPN classification. In *Int'l Conf. on Field-Programmable Technology*. 310–313.
[10] Donald E. Knuth. 2011. *The Art of Computer Programming*. Vol. 4A. Addison-Wesley, Upper Saddle River, New Jersey.
[11] Donald E. Knuth. 2015. *The Art of Computer Programming, Volume 4, Fascicle 6: Satisfiability*. Addison-Wesley, Reading, Massachusetts.
[12] Arist Kojevnikov, Alexander S. Kulikov, and Grigory Yaroslavtsev. 2009. Finding efficient circuits using SAT-solvers. In *Theory and Applications of Satisfiability Testing*. 32–44.
[13] Alan Mishchenko. 2001. *An Approach to Disjoint-Support Decomposition of Logic Functions*. Technical Report. Portland State University.
[14] Alan Mishchenko, Satrajit Chatterjee, and Robert K. Brayton. 2007. Improvements to Technology Mapping for LUT-Based FPGAs. *IEEE Trans. on CAD of Integrated Circuits and Systems* 26, 2 (2007), 240–253.
[15] John P. Roth and Richard M. Karp. 1962. Minimization Over Boolean Graphs. *IBM Journal of Research and Development* 6, 2 (1962), 227–238.
[16] Mathias Soeken, Luca Amarù, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli. 2017. Exact Synthesis of Majority-Inverter Graphs and Its Applications. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2017). https://doi.org/10.1109/TCAD.2017.2664059
[17] Mathias Soeken, Giovanni De Micheli, and Alan Mishchenko. 2017. Busy Man's Synthesis : Combinational Delay Optimization With SAT. In *Design Automation and Test in Europe*.