2023 Design, Automation & Test in Europe Conference (DATE 2023)

Exact Synthesis Based on Semi-Tensor Product Circuit Solver

Hongyang Pan and Zhufei Chu*
Faculty of Electrical Engineering & Computer Science, Ningbo University, Ningbo, China
*Email: chuzhufei@nbu.edu.cn

***Abstract*—In logic synthesis, Boolean satisfiability (SAT) is widely used as a reasoning engine, especially for exact synthesis. By representing input formulas as logic circuits instead of conjunction normal forms (CNFs) as in off-the-shelf CNF-based SAT solvers, circuit-based SAT solvers enable decoding after solution to be easier. An exact synthesis method based on a semi-tensor product (STP) circuit solver is presented in this paper. As opposed to other SAT-based exact synthesis algorithms, synthesized Boolean functions are encoded into STP canonical forms and can be solved by STP-based circuit SAT solver in our method. It can also obtain all optimal solutions in one pass. In particular, all solutions are expressed as 2-lookup tables (LUTs), rather than homogeneous logic representations. Hence, different costs can be considered when selecting the optimal circuit. In experiments, we demonstrate that our method accelerates the runtime up to 225.6X while reducing timeout instances by up to 88%.**

***Index Terms*—exact synthesis, semi-tensor product of matrices, Boolean satisfiability, circuit satisfiability**

I. Introduction

Boolean satisfiability (SAT) is the first problem that was proven to be non-deterministic polynomial (NP)-complete. All solutions SAT (AllSAT) is a variant of the SAT problem that consists of determining all satisfying assignments for a given propositional logic formula. The SAT formula has been solved by a number of efficient SAT solvers [1], and thus has broad practical applications. SAT has been used in logic synthesis to synthesize optimum Boolean chains, also known as exact synthesis, where network optimality is determined by some cost function [2]. Specifically, finding the network with the fewest nodes or logic levels is an example. Exact synthesis can be made significantly easier by SAT-based implementations. The reason for this is that SAT-based exact synthesis exploits the solution space implicitly rather than explicitly enumerating all candidates. There are attempts, such as the development of alternative *conjunctive normal form* (CNF) encoding, demonstrating the quantitative differences between CNF encoding [3], and integrating *directed acyclic graph* (DAG) topology families into SAT solver [4], to speed up the exact synthesis.

In recent years, circuit-based SAT solvers have achieved significant progress [5]. Circuit-based SAT represents an input formula as a logic circuit without converting it into CNF so it does not lose the topology of the circuit. An evaluation of assignment satisfaction can be made by relying on structural information, such as circuit connectivity. In spite of the popularity of CNF-based encoding, translating from instance to CNF increases encoding size [6]. A circuit-based encoding can also reduce the number of variables in the formulation, making decoding simpler. Therefore, increased compute power, coupled with high-speed circuit-based SAT solvers, could lead to more efficient algorithms for exact synthesis.

In this paper, we propose an exact synthesis algorithm based on *semi-tensor product* (STP) circuit solver. The STP method works on matrices. A logic matrix can be used to define the Boolean variables in order to prove the basic properties of logic using the STP method [7]. By combining logical reasoning and SAT solving, we propose a novel circuit-based SAT solver that solves the exact synthesis problem. The main contributions of this paper are as follows.

- We define logic matrices, which convert logical reasoning into mathematical computation and preserve the topological information between circuits, as primitives in the logic network to solve the circuit SAT problem.
- Synthesized Boolean functions are encoded into STP canonical forms and can be solved by STP-based circuit SAT solver. We use the DAG topology families [4] to reduce the synthesis runtime as well as the solution spaces. The STP-based exact synthesis can also generate all optimal solutions under current constraints by one pass.
- Synthesized Boolean functions are represented as STP canonical forms. These matrices are factored into small scale logic matrices and assigned to the vertices of DAGs to realize the specified Boolean function. They are then used as input to circuit-based SAT solvers.
- The proposed exact synthesis method is implemented in C++ on top of the logic synthesis framework ALSO[1], in which the source codes are publicly available. Compared with state-of-the-art exact synthesis algorithms, our method has a CPU time reduction of up to 225.6x and reduces the number of timeouts by up to 88%.

II. Preliminaries

A. Semi-Tensor Product of Matrices and Its Logical Reasoning

This subsection gives a brief review of the STP calculation of matrices and its logical reasoning. We refer the reader to [8] for more details. The real matrices with $m \times n$ dimensions are represented by $M^{m\times n}$. Consider two matrices $X \in M^{m\times n}$ and $Y \in M^{p\times q}$, the STP can produce matrices in any dimension.

Definition 1. *Let $X \in M^{m\times n}$ and $Y \in M^{p\times q}$, the STP of X and Y, denoted by $X \ltimes Y$, is defined as*

This work was supported in part by the National Natural Science Foundation of China under Grants 61871242 and 62274100, and in part by the State Key Laboratory of ASIC & System under Grant 2021KF008.

[1]Chu Z. ALSO: Advanced logic synthesis and optimization tool. https://github.com/nbulsi/also, 2022.

978-3-9819263-7-8/DATE23/© 2023 EDAA

$$X \ltimes Y = (X \otimes I_{t/n}) \cdot (Y \otimes I_{t/p}),$$

where I represents identity matrix, t is the least common multiple (lcm) of n and p, and $\otimes$ is Kronecker product of two arbitrary dimensional matrices [9].

Property 1. *The STP of matrices can realize matrix swapping. Let X be a matrix with any dimensions, if $Z_r \in M^{1\times t}$ is a row vector, then $X \ltimes Z_r = Z_r \ltimes (I_t \otimes X)$. In contrast, if $Z_c \in M^{t\times 1}$ is a column vector, then $Z_c \ltimes X = (I_t \otimes X) \ltimes Z_c$.*

The matrix form of logic formulas can be used to describe logic representations in general. It is also useful for solving SAT problems. We refer to the matrix product as the STP in this paper and omit the symbol "$\ltimes$" hereinafter. First, we denote the set of Boolean variables S_V.

$$S_V : \left\{ True = \begin{bmatrix}1\\0\end{bmatrix}, False = \begin{bmatrix}0\\1\end{bmatrix}. \right\} \tag{1}$$

Definition 2. *A 2×2^n matrix is called a logic matrix if all its columns are elements in S_V.*

Definition 3. *A logic matrix M_σ in which columns are consistent with the truth table (it is read from right to left) of a logic operation σ is called the structural matrix of σ.*

Example 1. *Assume $a, b \in S_V$ and σ is an Boolean operator. For unary operator "not", the structural matrix is $M_{n(\neg)} = \begin{bmatrix}0&1\\1&0\end{bmatrix}$. The inversion of variable a can be converted to matrices multiplication as $\bar{a} = M_n a$. For binary operators, it can be converted as*

$$a\ \sigma\ b = M_\sigma ab. \tag{2}$$

Therefore, any Boolean function can be converted into its STP form by their structural matrices.

Example 2. *Prove $a \to b = \bar{a} \vee b$ using STP forms.*

Proof. The structural matrices of the used Boolean operators are shown as follows,

$$M_{d(\vee)} = \begin{bmatrix}1&1&1&0\\0&0&0&1\end{bmatrix},\ M_{i(\to)} = \begin{bmatrix}1&0&1&1\\0&1&0&0\end{bmatrix}.$$

The STP form of left hand side is $M_i ab$, while the right hand side is $M_d(M_n a)b$.

$$M_d M_n = \begin{bmatrix}1&1&1&0\\0&0&0&1\end{bmatrix}\begin{bmatrix}0&1\\1&0\end{bmatrix} = \begin{bmatrix}1&0&1&1\\0&1&0&0\end{bmatrix} = M_i.$$

Hence, the identity holds. $\square$

Logic identities can be proved easily using structure matrices of Boolean operators and STP properties. Moreover, in the case of multiple logic variable multiplies such as $a \cdot a = a^2$, we can consider a variable power-reducing matrix $M_r \in M^{4\times 2}$ for further processing. M_r is defined as

$$M_r = \begin{bmatrix}1&0\\0&0\\0&0\\0&1\end{bmatrix}. \tag{3}$$

Next, the variable swapping matrix $M_w \in M^{4\times 4}$ is used to sort the logic variables in order. M_w is defined as

$$M_w = \begin{bmatrix}1&0&0&0\\0&0&1&0\\0&1&0&0\\0&0&0&1\end{bmatrix}. \tag{4}$$

Example 3. *Assume a and b are logic variables. We can implement $a^2 = M_r a$ with the variable power-reducing matrix M_r or $M_w ba = ab$ with the variable swapping matrix M_w.*

A key aspect of Boolean function manipulation is the canonical form, since these functions can be functionally equivalently represented in several different logic realizations. A canonical form is also available for STP.

Property 2. *Any logic expression $\Phi(x_1, \ldots, x_n)$ with Boolean variables $x_1, \ldots, x_n \in S_V$ can be calculated into a canonical form as*

$$\Phi(x_1, \ldots, x_n) = M_\Phi x_1 \ldots x_n,$$

where $M_\Phi \in M^{2\times 2^n}$. The calculation needs Boolean variable power-reducing and matrix swapping (see Property 1, (3), and (4)).

Example 4. *We revisit an example in [10] to explain the STP calculation process. There are three persons a, b, and c. They are either honest or liar, suppose a liar always said a lie and the honest man always told the truth. Person a said that person b is a liar, person b said person c is a liar, and person c said that both a and b are liars. Who is/are the liar(s)?*

First, we define Boolean variable a to indicate person a is honest. Thus $\neg a$ means a is a liar. The definitions also work for Boolean variables b and c. The statements result the logic expression

$$\Phi(a, b, c) = (a \leftrightarrow \neg b) \wedge (b \leftrightarrow \neg c) \wedge (c \leftrightarrow \neg a \wedge \neg b). \tag{5}$$

The STP form of (5) is

$$\begin{aligned}\Phi(a, b, c) &= M_\Phi abc\\ &= M_c^2 (M_e a M_n b)(M_e b M_n c)(M_e c M_c M_n a M_n b),\end{aligned}$$

where M_c and M_e respectively represent the structural matrices of conjunction ($\wedge$) and equivalence ($\leftrightarrow$). Then, converting the STP form of logic expression into the canonical form as

$$\Phi(a, b, c) = M_\Phi abc = \begin{bmatrix}0&0&0&0&0&1&0&0\\1&1&1&1&1&0&1&1\end{bmatrix} abc.$$

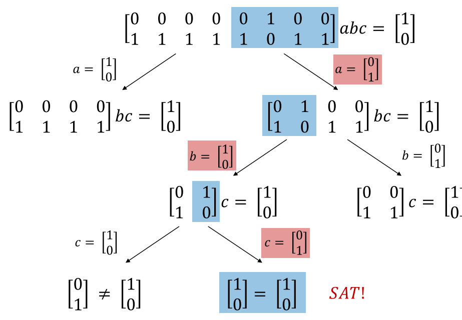

$\begin{bmatrix}0&0&0&0&0&1&0&0\\1&1&1&1&1&0&1&1\end{bmatrix} abc = \begin{bmatrix}1\\0\end{bmatrix}$
$a = \begin{bmatrix}1\\0\end{bmatrix}$
$a = \begin{bmatrix}0\\1\end{bmatrix}$
$\begin{bmatrix}0&0&0&0\\1&1&1&1\end{bmatrix} bc = \begin{bmatrix}1\\0\end{bmatrix}$
$\begin{bmatrix}0&1&0&0\\1&0&1&1\end{bmatrix} bc = \begin{bmatrix}1\\0\end{bmatrix}$
$b = \begin{bmatrix}1\\0\end{bmatrix}$
$b = \begin{bmatrix}0\\1\end{bmatrix}$
$\begin{bmatrix}0&1\\1&0\end{bmatrix} c = \begin{bmatrix}1\\0\end{bmatrix}$
$\begin{bmatrix}0&0\\1&1\end{bmatrix} c = \begin{bmatrix}1\\0\end{bmatrix}$
$c = \begin{bmatrix}1\\0\end{bmatrix}$
$c = \begin{bmatrix}0\\1\end{bmatrix}$
$\begin{bmatrix}0\\1\end{bmatrix} \neq \begin{bmatrix}1\\0\end{bmatrix}$
$\begin{bmatrix}1\\0\end{bmatrix} = \begin{bmatrix}1\\0\end{bmatrix}$
SAT!

Fig. 1: The STP calculation process.

The SAT problem is to determine in which case $\Phi(a, b, c) = \begin{bmatrix}1\\0\end{bmatrix}$. The calculation is straightforward since we need to find appropriate values in S_v for each logic variable, shown in Fig. 1. We can determine the logic values of a, b, and c in sequence. Each time we assign a logic value, the dimensions of M_Φ is reduced from 2×2^n to $2 \times 2^{n-1}$. In fact, we need to *extract the column* $\begin{bmatrix}1\\0\end{bmatrix}$ *in* M_Φ *to satisfy the SAT problem. If the assignment makes* M_Φ *contains no* $\begin{bmatrix}1\\0\end{bmatrix}$ *column, that means no satisfiability results can be found and would backtrack to other assignments. The STP is well suited for solving SAT and AllSAT problems [11] by extracting the column* $\begin{bmatrix}1\\0\end{bmatrix}$ *in the canonical form. Hence, the only solution is*

$$a = \begin{bmatrix}0\\1\end{bmatrix}, b = \begin{bmatrix}1\\0\end{bmatrix}, c = \begin{bmatrix}0\\1\end{bmatrix}.$$

Thus, b is honest.

B. Boolean Chains

Boolean chains may be viewed as a precise formal model of the concept of multi-level logic networks. The definition of Boolean chains is originally introduced by Knuth [12]. A Boolean chain is a DAG in which every internal vertex has a corresponding k-input Boolean operator $\phi : \mathbb{B}^k \to \mathbb{B}$. The set of allowed operators is denoted by $\mathcal{B}$. Let $f = (f_1, \ldots, f_m)$ be a multiple-output Boolean function, such that $f : \mathbb{B}^n \to \mathbb{B}^m$ and the functions $f_1, \ldots, f_m$ are defined over common support $x_1, \ldots, x_n$. Then, for $k \geqslant 1$ and a set $\mathcal{B}$, a k-input operator Boolean chain is a sequence $(x_{n+1}, \ldots, x_{n+r})$, where

$$x_i = \phi_i(x_{j(i,1)}, \ldots, x_{j(i,k)}) \qquad (n+1 \leqslant i \leqslant n+r)$$

such that $\phi_i \in \mathcal{B}, 1 \leqslant j(i,\cdot) < i$, and for all $1 \leqslant k \leqslant m$, either $f_k(x_1, \ldots, x_n) = x_{l(k)}$ or $f_k(x_1, \ldots, x_n) = \bar{x}_{l(k)}$, where $0 \leqslant l(k) \leqslant n+r$, and $x_0 = 0$ the constant zero input.

III. STP-based Exact Synthesis

The runtime of SAT-based exact synthesis has always been unpredictable and potentially slow. When finding optimum Boolean chains in [12], the SAT solver can solve faster if it can perform the following two tasks.

1) To realize the specified Boolean function, find DAG structures for the Boolean chain, and assign Boolean operators to their vertices.
2) Solve the SAT problem with circuit information directly instead of encoding CNF.

Based on these two tasks, we aim to generate possible optimum Boolean chains in DAG structures, and take them as inputs of circuit AllSAT solver without encoding to CNF.

The input of our proposed exact synthesis algorithm is a synthesized Boolean function f and the output is a set of all optimum Boolean chains candidates S_o. The algorithm proceeds as in the following.

(i) Initialize the constraint n, which is equal to the input number of f minus one, to specific the number of logic gates in Boolean chain.

(ii) Generate all satisfying the constraint possible DAG $pDAGs$, where it will update the constraint n (**Section III-A**).

(iii) Encode all possible optimum Boolean chain candidates pBC by STP-based encoding (**Section III-B**). If there are no pBC, go to (i) and update the constraint $n = n + 1$.

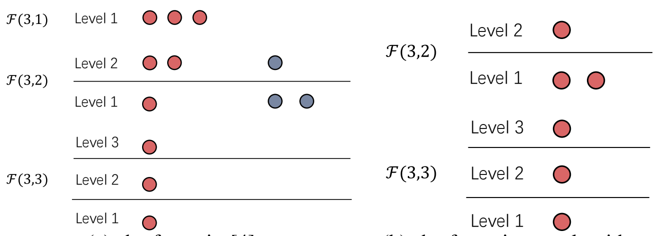
(iv) If the pBC checked by AllSAT solver is satisfying and the simulation of all satisfying solutions is f, the pBC is the optimal solution and saved into S_o (**Section III-C**). Otherwise, go to step (i) and update $n = n + 1$.

A. DAG Topology Families

Topology-based exact synthesis can reduce the search space of the SAT solver by providing additional constraints. In order to obtain all possible DAGs, $pDAG$s, satisfying the Boolean function f quickly, our method is based on existing CNF-based SAT encoding[2]. Given two integers k and l ($1 \leq l \leq k$), a *Boolean fence* is a partition of k nodes over l levels, where each level contains at least one node. The set of all *Boolean fence family* is denoted by $\mathcal{F}(k,l)$ and $\mathcal{F}_k$ denotes the $\mathcal{F}(k,l)$ of k nodes

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

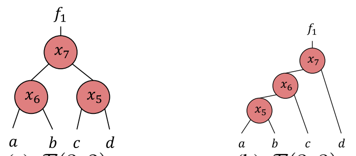
Fig. 2(a) shows the fences of $\mathcal{F}_3$ in [4].



(a) the fence in [4] (b) the fence in our algorithm

Fig. 2: Illustration of fence $\mathcal{F}_3$

In order to reduce the synthesis runtime as well as the solution space, we use some additional constraints. Since all the functions have a single output, we will prune the DAG in $\mathcal{F}_k$ with more than one node at the top. There should be no more than two nodes between a higher logic level and each lower logic level. This is because we want to synthesize operators with only two inputs in Boolean chains. Fig. 2(b) shows the fences of $\mathcal{F}_3$ in our algorithm. Every fence corresponds to a family of DAGs with the same distribution of nodes across levels. We draw an edge here between the first node on each level in a fence for visualization purposes. Therefore, we can generate the DAGs with connectivity information based on $\mathcal{F}_k$. In addition, we use the *negation-permutation-negation* (NPN) classification to reduce the size of all valid DAG candidates. Two Boolean functions are NPN-equivalent if one can be obtained from the other by negating (i.e., complementing) and permuting inputs, and negating the output [13]. Fig. 3 shows all valid DAGs with connectivity information of $\mathcal{F}_3$.



(a) $\mathcal{F}(3,2)$ (b) $\mathcal{F}(3,3)$

Fig. 3: Illustration of all valid DAGs of $\mathcal{F}_3$.

B. Encode Boolean Chain Candidates

In this subsection, we assign Boolean operators to the vertices of the $pDAG$s. Instead of using SAT solver directly, we adopt STP theory for possible matrix factorization of the given synthesized Boolean function f into logic matrices. The DAGs which can not realize the Boolean function f are pruned.

[2]Haaswijk W. PERCY: A header-only exact synthesis library. https://github.com/whaaswijk/percy, 2019.

First, we consider the f as a STP canonical form. Therefore, we define the logic matrix $M_\Phi \in M^{2\times 2^n}$ of f as

$$M_\Phi = \begin{bmatrix} M_x & M_y & M_z & M_v \\ \bar{M}_x & \bar{M}_y & \bar{M}_z & \bar{M}_v \end{bmatrix}, \tag{6}$$

where M_Φ is divided into four parts, and M_x, M_y, M_z, and $M_v \in M^{2\times 2^{n-2}}$ are also logic matrices (see Definition 2). We use the logic matrix M_Φ in (6) as a known canonical form hereinafter.

As opposed to the calculation of canonical form in Property 1, we propose a STP-based matrix factorization algorithm to decompose the known canonical form, defined as

$$M_\Phi \xrightarrow[decomposition]{matrix\text{-}factorization} M_{\Phi_a} M_{\Phi_b}.$$

The M_Φ can be factored to M_{Φ_a} and M_{Φ_b} if and only if there are two unique quartering parts of M_Φ, which can be verified by Definition 1. Otherwise, the corresponding $pDAG$ can not realize the target Boolean function f.

Example 5. *1. If* $M_\Phi = \begin{bmatrix} M_a & M_b & M_a & M_a \\ \bar{M}_a & \bar{M}_b & \bar{M}_a & \bar{M}_a \end{bmatrix}$*. The* M_Φ *can be factored to* M_{Φ_1} *and* M_{Φ_2} *as*

$$M_{\Phi_1} = \begin{bmatrix} M_a & M_b \\ \bar{M}_a & \bar{M}_b \end{bmatrix},\ M_{\Phi_2} = \begin{bmatrix} 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \end{bmatrix};$$

or

$$M_{\Phi_1} = \begin{bmatrix} M_b & M_a \\ \bar{M}_b & \bar{M}_a \end{bmatrix},\ M_{\Phi_2} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 \end{bmatrix}.$$

2. If $M_\Phi = \begin{bmatrix} M_a & M_b & M_c & M_a \\ \bar{M}_a & \bar{M}_b & \bar{M}_c & \bar{M}_a \end{bmatrix}$*. There are three unique quartering parts, that is,* $\begin{bmatrix} M_a \\ \bar{M}_a \end{bmatrix}$, $\begin{bmatrix} M_b \\ \bar{M}_b \end{bmatrix}$*, and* $\begin{bmatrix} M_c \\ \bar{M}_c \end{bmatrix}$*, in* M_Φ*. Therefore, the* M_Φ *can not be factored.*

In addition, if the matrix is swapped m times, $n = 2^m$ in I_n (see Property 1). Therefore, when the matrices are swapped, which means M_Φ includes I_n, M_Φ will be divided in half. Then, M_Φ can be factored if and only if there are two unique quartering parts of each half-M_Φ, and the position of two unique quartering parts of each half-M_Φ must be consistent.

Example 6. *1. If* M_Φ *is divided into eight parts, denoted by* $\begin{bmatrix} M_a & M_b & M_b & M_b & M_c & M_d & M_d & M_d \\ \bar{M}_a & \bar{M}_b & \bar{M}_b & \bar{M}_b & \bar{M}_c & \bar{M}_d & \bar{M}_d & \bar{M}_d \end{bmatrix}$*, and we have*

$$\begin{aligned} M_\Phi &= M_{\Phi_3}(I_2 \otimes M_{\Phi_4}) \\ &= \begin{bmatrix} M_a & M_b & M_b & M_b & M_c & M_d & M_d & M_d \\ \bar{M}_a & \bar{M}_b & \bar{M}_b & \bar{M}_b & \bar{M}_c & \bar{M}_d & \bar{M}_d & \bar{M}_d \end{bmatrix}, \end{aligned}$$

M_Φ *can be factored to* M_{Φ_3} *and* M_{Φ_4} *as*

$$M_{\Phi_3} = \begin{bmatrix} M_a & M_b & M_c & M_d \\ \bar{M}_a & \bar{M}_b & \bar{M}_c & \bar{M}_d \end{bmatrix},\ M_{\Phi_4} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \end{bmatrix};$$

or

$$M_{\Phi_3} = \begin{bmatrix} M_b & M_a & M_d & M_c \\ \bar{M}_b & \bar{M}_a & \bar{M}_d & \bar{M}_c \end{bmatrix},\ M_{\Phi_4} = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix}.$$

2. If M_Φ *is divided into eight parts, denoted by* $\begin{bmatrix} M_a & M_b & M_b & M_b & M_c & M_c & M_d & M_d \\ \bar{M}_a & \bar{M}_b & \bar{M}_b & \bar{M}_b & \bar{M}_c & \bar{M}_c & \bar{M}_d & \bar{M}_d \end{bmatrix}$*, and we have* $M_\Phi = M_{\Phi_3}(I_2 \otimes M_{\Phi_4})$*. Although there are only two unique quartering parts of each half-*M_Φ*, the position of two unique quartering parts of each half-*M_Φ *is different. Therefore, the* M_Φ *can not be factored.*

If there are matrix swapping or variable power-reducing in M_Φ, the M_Φ can be also factored (see Section (3) and (4)).

Property 3. *1. If* $M_\Phi = M_{\Phi_5} M_r = \begin{bmatrix} M_a & M_b & M_c & M_d \\ \bar{M}_a & \bar{M}_b & \bar{M}_c & \bar{M}_d \end{bmatrix}$*. The matrix* M_{Φ_5} *can be factored as*

$$M_{\Phi_5} \begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} M_a & M_b & M_c & M_d \\ \bar{M}_a & \bar{M}_b & \bar{M}_c & \bar{M}_d \end{bmatrix}$$

$$M_{\Phi_5} = \begin{bmatrix} M_a & M_b & x & x & x & x & M_c & M_d \\ \bar{M}_a & \bar{M}_b & \bar{x} & \bar{x} & \bar{x} & \bar{x} & \bar{M}_c & \bar{M}_d \end{bmatrix},$$

where the 'x' represents 0 or 1. As a result, the dimensions of M_Φ *go from* $(2 \times n)$ *to* $(2 \times 2n)$*.*

2. If $M_\Phi = M_{\Phi_6} M_w = \begin{bmatrix} M_a & M_b & M_c & M_d \\ \bar{M}_a & \bar{M}_b & \bar{M}_c & \bar{M}_d \end{bmatrix}$*. The matrix* M_{Φ_6} *can be factored as*

$$M_{\Phi_6} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} M_a & M_b & M_c & M_d \\ \bar{M}_a & \bar{M}_b & \bar{M}_c & \bar{M}_d \end{bmatrix}$$

$$M_{\Phi_6} = \begin{bmatrix} M_a & M_c & M_b & M_d \\ \bar{M}_a & \bar{M}_c & \bar{M}_b & \bar{M}_d \end{bmatrix},$$

where the second and third parts of M_Φ *are swapped.*

Similarly, if there are matrix swapping in M_Φ, the M_Φ is first divided in half and then the half-M_Φs are swapped or power-reduced respectively.

Property 4. *1. If* $M_\Phi = M_{\Phi_7}(I_2 \otimes M_r) = \begin{bmatrix} M_a & M_b & M_c & M_d \\ \bar{M}_a & \bar{M}_b & \bar{M}_c & \bar{M}_d \end{bmatrix}$*. The matrix* M_{Φ_7} *can be factored as*

$$M_{\Phi_7} \begin{bmatrix} M_r & \boldsymbol{0} \\ \boldsymbol{0} & M_r \end{bmatrix} = \begin{bmatrix} M_a & M_b & M_c & M_d \\ \bar{M}_a & \bar{M}_b & \bar{M}_c & \bar{M}_d \end{bmatrix}$$

$$M_{\Phi_7} = \begin{bmatrix} M_a & x & x & M_b & M_c & x & x & M_d \\ \bar{M}_a & \bar{x} & \bar{x} & \bar{M}_b & \bar{M}_c & \bar{x} & \bar{x} & \bar{M}_d \end{bmatrix},$$

where $\boldsymbol{0}$ *is the 0 matrix with same dimensions as* M_r*.*

2. If the logic matrix M_Φ *is divided into eight parts, denoted by* $\begin{bmatrix} M_a & M_b & M_c & M_d & M_e & M_f & M_g & M_h \\ \bar{M}_a & \bar{M}_b & \bar{M}_c & \bar{M}_d & \bar{M}_e & \bar{M}_f & \bar{M}_g & \bar{M}_h \end{bmatrix}$*, and* $M_\Phi = M_{\Phi_8}(I_2 \otimes M_w)$*. The matrix* M_{Φ_8} *can be factored as*

$$M_{\Phi_8} \begin{bmatrix} M_w & \boldsymbol{0} \\ \boldsymbol{0} & M_w \end{bmatrix} = \begin{bmatrix} M_a & M_b & M_c & M_d & M_e & M_f & M_g & M_h \\ \bar{M}_a & \bar{M}_b & \bar{M}_c & \bar{M}_d & \bar{M}_e & \bar{M}_f & \bar{M}_g & \bar{M}_h \end{bmatrix}$$

$$M_{\Phi_8} = \begin{bmatrix} M_a & M_c & M_b & M_d & M_e & M_g & M_f & M_h \\ \bar{M}_a & \bar{M}_c & \bar{M}_b & \bar{M}_d & \bar{M}_e & \bar{M}_g & \bar{M}_f & \bar{M}_h \end{bmatrix},$$

where $\boldsymbol{0}$ *is the 0 matrix with same dimensions as* M_w*, and the second and third parts of each half-*M_Φ *are swapped.*

Using these properties, we can get all possible optimum Boolean chain candidates of DAGs under the constraint n. We present a simple example to depict it.

Example 7. *In Fig. 3(a), there is a DAG with four inputs, and the synthesized Boolean function is* `0x8ff8` *in hexadecimal representation.*

$$\begin{aligned}\Phi(a,b,c) &= M_7 M_6 ab M_5 cd = M_7 M_6 (I_4 \otimes M_5) abcd \\ &= \begin{bmatrix}1&0&0&0&1&1&1&1&1&1&1&1&1&0&0&0\\0&1&1&1&0&0&0&0&0&0&0&0&0&1&1&1\end{bmatrix} abcd\end{aligned}$$

$$M_7 M_6 (I_4 \otimes M_5) = \begin{bmatrix}1&0&0&0&1&1&1&1&1&1&1&1&1&0&0&0\\0&1&1&1&0&0&0&0&0&0&0&0&0&1&1&1\end{bmatrix} \quad (7)$$

The matrix M_7, M_6, M_5 in (7) can be factored as

$$M_7 = \begin{bmatrix}1&1&1&0\\0&0&0&1\end{bmatrix}, M_6 = \begin{bmatrix}1&0&0&0\\0&1&1&1\end{bmatrix}, M_5 = \begin{bmatrix}0&1&1&0\\1&0&0&1\end{bmatrix}$$

or

$$M_7 = \begin{bmatrix}0&1&1&1\\1&0&0&0\end{bmatrix}, M_6 = \begin{bmatrix}0&1&1&1\\1&0&0&0\end{bmatrix}, M_5 = \begin{bmatrix}1&0&0&1\\0&1&1&0\end{bmatrix}$$

Thus, all Boolean chain candidates of this DAG are,

$$\begin{cases} x_7 = \mathtt{0xe}(x_5, x_6), & x_6 = \mathtt{0x8}(a,b), \quad x_5 = \mathtt{0x6}(c,d); \\ x_7 = \mathtt{0x7}(x_5, x_6), & x_6 = \mathtt{0x7}(a,b), \quad x_5 = \mathtt{0x9}(c,d). \end{cases}$$

As for the correctness of solutions, we can convert the candidates to their corresponding STP form, and compute the matrices again. However, a power-reducing matrix will result in x (0 or 1) in Boolean operators, which will cause error in calculations. Therefore, we need a more efficient engine to verify the correctness of all possible optimum Boolean chain candidates, that is, a circuit-based SAT solver.

C. Circuit-based AllSAT Solver

In circuit-based solver, it determines whether an assignment to the circuit variables causes the circuit be $True$, that is, determines whether the assignments of primary inputs (PIs) can be all primary outputs (POs) true. In STP, the truth table of a logic operator is consistent with its logic matrix. Thus, we use the look-up tables (LUTs) as input. The circuit-based SAT solving algorithm is shown in Algorithm 1.

Algorithm 1: STP-Based Circuit Solving Algorithm.

```
   Input: the LUTs $L$
   Output: a set of solutions ($\mathbb{S}$)
1  $\#PIs, \#POs, T \leftarrow 1$;
2  $\mathbb{S} \leftarrow \{(\underbrace{-,...,-}_{\#PIs})\}$;
3  for $i = 1$ to $\#POs$ do
4      $\mathbb{S}_i \leftarrow$ TRAVERSE($PO_i$, T);
5      $\mathbb{S}^* \leftarrow$ MERGE($\mathbb{S}_i$, $\mathbb{S}$);
6      $\mathbb{S} \leftarrow \mathbb{S}^*$;
7  end
8  if size($\mathbb{S}$) then
9      return $\mathbb{S}$ & SAT;
10 else
11     return UNSAT;
12 end
```

(i) *Initialization*. We derive the number of POs (*#POs*) and the number of PIs (*#PIs*). The variable *T* is initialized to 1 to ensure that $po_i \in POs$ have satisfiability assignments (line 1). The initial solution $\mathbb{S}$ is set to a *#PIs*-length string with each index value of '-' which means *Unassigned* (line 2).

(ii) *Calculation*. The calculation is recursively operated for each $po_i \in PO$ until its children nodes reach *PIs*, shown in Algorithm 2. Given a circuit *node* and the target assignment *T*, we first get the children of *node* and convert the logic expression to its STP form M_i (lines 5-6). New *T*s will be returned for the children nodes by the STP calculation in order to continue the processing (line 7). Next, the traverse function continues to work on the two children nodes (lines 8-9).

(iii) *Judging*. The calculation process continues until all $po_i \in POs$ have been traversed. The solver returns SAT and the set of all solutions $\mathbb{S}$ if it finds at least one solution in $\mathbb{S}$, otherwise returns UNSAT. Then, we simulate all solutions of $\mathbb{S}$ to a Boolean function f_s and check whether f_s is equal to target Boolean function f. If $f_s == f$, this Boolean chain is a optimum Boolean chain.

Algorithm 2: Recursive Traverse Function.

```
1  Traverse($node$, $T$);
2  if $node \in PI$ then
3      return $T$;
4  end
5  $Children \leftarrow$ Get_Children( $node$ );
6  $M_i \leftarrow$ Convert_to_STP_form($Children$);
7  $Ts \leftarrow$ STP_calculation($M_i$, $T$);
8  Traverse($Children[0]$, $Ts[0]$);
9  Traverse($Children[1]$, $Ts[1]$);
```

Example 8. *We use the Boolean chain generated by Example 7. There are four PIs (a,b,c,d) and one PO (f_1). The initial solution is a 4-length string of the form $\mathbb{S} = (-,-,-,-)$. The first Target of f_1 (T_0) is a satisfying assignment "1" and we can update the Target of x_7 (T_1 = "1"). The structural matrix of x_7 is $\begin{bmatrix}1&1&1&0\\0&0&0&1\end{bmatrix}$, the possible satisfying assignments of x_5 and x_6 in x_7 is "11", "10", and "01". Then we can update the Target of x_5 (T_2) and x_6 (T_3) with (T_2 = "1", T_3 = "1"), (T_2 = "1", T_3 = "0"), and (T_2 = "0", T_3 = "1").*

$$\begin{aligned} f_1 &= \begin{bmatrix}1&0\\0&1\end{bmatrix} x_7 && (T_0 = 1) \\ x_7 &= \begin{bmatrix}1&1&1&0\\0&0&0&1\end{bmatrix} x_6 x_5 && (T_1 = 1) \\ x_6 &= \begin{bmatrix}1&0&0&0\\0&1&1&1\end{bmatrix} ab && (T_3 = 1,0,1) \\ x_5 &= \begin{bmatrix}0&1&1&0\\1&0&0&1\end{bmatrix} cd && (T_2 = 1,1,0) \end{aligned}$$

Again, the structural matrices of x_6 and x_5 are $\begin{bmatrix}1&0&0&0\\0&1&1&1\end{bmatrix}$ and $\begin{bmatrix}0&1&1&0\\1&0&0&1\end{bmatrix}$. The corresponding Targets T_3 and T_2 are ("1","1"), ("1","0"), and ("0","1"). Thus, there are ten possible satisfying assignments of a,b,c,d, which are ("1,1,0,1"), ("1,1,1,0"), ("0,1,0,1"), ("0,1,1,0"), ("1,0,0,1"), ("1,0,1,0"), ("0,0,0,1"), ("0,0,1,0"), ("1,1,1,1"), and ("1,1,0,0"). We simulate these assignments to a Boolean function $f_s = 0x8ff8$, which is equal to the target Boolean function f. Hence, the Boolean chain candidate in Example 7 is the optimal solution.

IV. Experimental results

All experiments are performed on a 2.40 GHz Intel(R) Xeon(R) Silver 4210R CPU with 64 GB of main memory. The results are verified by simulating the truth tables to ensure correctness.

TABLE I: Experimental Results

Functions	BMS			FEN			ABC			STP				
	mean(s)	#t/o	#ok	mean(s)	#t/o	#ok	mean(s)	#t/o	#ok	Total(s)	mean(s)	#t/o	#ok	number
NPN4	0.235	0	222	0.208	0	222	0.167	0	222	3.264	**0.136**	0	222	**24**
FDSD6	0.085	0	1000	0.036	0	1000	0.072	0	1000	0.144	**0.012**	0	1000	**12**
FDSD8	10.602	0	100	4.492	0	100	2.337	0	100	2.256	**0.047**	0	100	**48**
PDSD6	38.395	256	744	19.340	128	872	51.161	256	744	1554.6	44.290	96	**904**	**64**
PDSD8	189.935	14	86	94.211	11	89	128.558	76	24	16795.2	117.475	9	**91**	**192**

To evaluate the performance of our method, we measure the runtime on the following collections of Boolean functions.

- *NPN4*: All 222 4-input NPN classes [15].
- *FDSD6*: 1000 fully-DSD decomposable 6-input functions that occur frequently in practical synthesis and technology mapping applications [16].
- *PDSD6*: 1000 common 6-input partially-DSD functions.
- *FDSD8*: 100 8-input fully-DSD functions.
- *PDSD8*: 100 8-input partially-DSD functions.

We compare three different approaches as:

1) *BMS [17]*: Baseline implementation of the SAT-based exact synthesis algorithm.
2) *FEN [3]*: The algorithm based on fence enumeration and the use of additional topological constraints.
3) *ABC*: The state-of-the-art reference implementation from the well known ABC[3] (command `lutexact`).
4) *STP*: Our proposed algorithm based on STP-based SAT solving and matrix factorization.

The experimental results are shown in Table I. For comparison, we give the mean solving time (***mean***), the number of instances that could not be solved in under 3 mins (***#t/o***), and the number of solved instances before the timeout (***#ok***). In addition, we also list the mean solving time of our algorithm (***Total***), the mean solving time of each solution (***mean***), and the average number of solutions (***number***). The ***#ok*** is an effective measure of a practical algorithm. There is obviously a preference for algorithms that can solve the most problems within a given runtime bound. For *NPN4* and *FDSD*, the CPU time can be reduced up to 225.6x, 95.6x, and 49.7x in comparison to *BMS*, *FEN*, and *ABC*, respectively. All algorithms find the solutions for all problem instances. The *STP* can obtain an average of 24, 12, and 48 solutions in a shorter time. In terms of *PDSD*, our algorithm has only achieved 1.2x acceleration compared to *ABC*. But we can solve more instances than the other three approaches before timing out. Moreover, The *STP* can also obtain an average of 64 and 192 solutions.

The results indicate that *STP* can significantly improve the number of instances that are solved while maintaining the CPU time. In fact, it dominates the other implementations with respect to the number of solved instances. The conventional SAT-based exact synthesis can only yield one solution, the *STP* can obtain all optimal Boolean chains of current topological constraints in one pass. And all solutions of the *STP* are not limited to specific Boolean operators. Therefore, the *STP* is more flexible and can select the most cost-effective implementation according to the actual design costs.

[3]Mishchenko A. ABC: System for sequential logic synthesis and formal verification. https://github.com/berkeley-abc/abc, 2022

V. Conclusion

An exact synthesis method based on the STP circuit solver is presented in this paper. In the off-the-shell SAT-based exact synthesis, logic network structure and node functionality must be encoded. However, the STP method provides advantages in logic matrix calculation and circuit connectivity. The main contributions are that we apply STP decomposition to prune invalid candidates and a STP-based circuit solver is proposed to verify the correctness of Boolean chain candidates. Results show that CPU time can be reduced by up to 225.6x and timeouts can be reduced by up to 88%.

References

[1] N. Eén, N. Sörensson. "An Extensible SAT-solver," in *Proc.Sixth International Conference on Theory & Applications of Satisfiability Testing*, pp. 502-518, 2004.

[2] H. Riener, W. Haaswijk, A. Mishchenko, et al. "On-the-fly and DAG-aware: Rewriting Boolean networks with exact synthesis," in *Proc.DATE*, pp. 1649-1654, 2019.

[3] W. Haaswijk, M. Soeken, A. Mishchenko, et al. "SAT-based exact synthesis: Encodings, topology families, and parallelism," in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(4): pp. 871-884, 2019.

[4] W. Haaswijk, A. Mishchenko, M. Soeken, et al. "SAT based exact synthesis using DAG topology families," in *Proc.DAC*, pp. 1-6, 2018.

[5] H. Zhang, J. Jiang, A. Mishchenko. "A circuit-based SAT solver for logic synthesis," in *Proc.ICCAD*, pp. 1-6, 2021.

[6] C. Thiffault, F. Bacchus, T. Walsh. "Solving Non-clausal Formulas with DPLL Search," in *Proc.Principles and Practice of Constraint Programming*, pp. 663-678, 2004.

[7] D. Cheng, H. Qi, A. Xue. "A Survey on Semi-Tensor Product of Matrices," in *Journal of Systems Science Complexity*, 20: pp. 304-322, 2007.

[8] D. Cheng, H. Qi, Y. Zhao. "An introduction to semi-tensor product of matrices and its applications," in *World Scientific*, 2012.

[9] Van Loan C F. "The ubiquitous Kronecker product," in *Journal of calculational and applied mathematics*, 123(1-2) pp. 85-100, 2000.

[10] D. Cheng. "On logic-based intelligent systems," in *Proc.International Conference on Control and Automation*, pp. 71-76, 2005.

[11] Ren X, Guo W, Mo Z, et al. "A Divide and Conquer Approach to All Solutions Satisfiability Problem," in *Proc.IEEE 4th International Conference on Computer and Communications (ICCC)*, pp. 2590-2595, 2018.

[12] D. E. Knuth, "The Art of Computer Programming", Volume 4A. Addis-onWesley, 2011.

[13] A. Petkovska, M. Soeken, G. De Micheli, et al. "Fast hierarchical NPN classification," in *Proc.FPL*, pp. 1-4, 2016.

[14] H. Pan, Z. Chu. "A Semi-Tensor Product Based All Solutions Boolean Satisfiability Solver," in *Journal of Computer Science and Technology*, 2022.

[15] W. Haaswijk, M. Soeken, L. Amarú, et al. "A novel basis for logic rewriting," in *Proc.ASP-DAC*, pp. 151-156, 2017.

[16] A. Mishchenko. "An approach to disjoint-support decomposition of logic functions," in *Elect. Eng. Comput. Sci.*, 2001.

[17] M. Soeken, G. De Micheli, A. Mishchenko. "Busy man's synthesis: Combinational delay optimization with SAT," in *Proc.DATE*, pp. 830-835, 2017.