Symmetry-Based Synthesis For Interpretable Boolean Evaluation

Andrea Costamagna
EPFL
Lausanne, Switzerland
andrea.costamagna@epfl.ch

Alan Mishchenko
UC Berkeley
Berkeley, California
alanmi@berkeley.edu

Satrajit Chatterjee
Kepler AI
Palo Alto, California
satrajit@gmail.com

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

***Abstract*—Efficient evaluation of Boolean functions is a fundamental problem in computer science, impacting computational complexity and hardware performance. A natural way to evaluate Boolean functions is using circuits composed of two-input operators. However, synthesizing minimum circuits for functions with more than 6 inputs is typically infeasible. This paper introduces an engine based on Edward's theory of symmetry-based remapping for synthesizing Boolean chains. The proposed engine can synthesize functions with up to 20 inputs within seconds, surpassing state-of-the-art tools that require extensive hyper-parameter tuning to handle similar functions and fail to scale beyond that. Additionally, it enhances the interpretability of Boolean chains, uncovering recursive substructures that facilitate optimality proofs and inform bit-wise manipulation algorithms.**

***Index Terms*—Boolean evaluation, combinational complexity, symmetric functions**

I. Introduction

BOOLEAN functions are fundamental to digital computation, serving as the building blocks for virtually all digital circuits. Compactly representing these functions as circuits is not only a theoretical challenge but also of significant practical importance, as it directly impacts resource usage and computation speed of such hardware components as multi-input adders. Despite decades of extensive research, finding optimal representations for functions with more than 6 inputs remains an elusive goal.

The primary challenge lies in determining a Boolean function's representation using the minimum number of binary logic operators, a measure known as the *combinational complexity* of the function. The process of finding this minimal representation is referred to as *exact synthesis*. Current state-of-the-art techniques encode exact synthesis as an instance of the *satisfiability* problem (SAT) [1], [2]. However, for functions whose circuits require more than 15 gates, SAT-based methods become impractical, leaving the search for close-to-optimal representations an open problem.

SAT-based synthesis employs a uniform encoding scheme for all Boolean functions, regardless of their specific characteristics. This raises a critical research question: *Can we achieve near-optimal solutions by tailoring the synthesis process to specific classes of functions?* We investigate this question by focusing on symmetric functions, which are particularly important for algorithms based on bit-wise manipulation [3] and in the context of binarized neural networks [4].

This research was supported in part by the SRC Contract 3173.001, "Standardizing Boolean transforms to improve the quality and runtime of CAD tools," and in part by Synopsys.

The present work targets synthesis of single-output Boolean functions, particularly those that can be effectively manipulated using truth tables but are beyond the practical reach of SAT-based methods, namely, functions with up to 20 inputs. By narrowing our focus to this class of functions, we demonstrate that specialized techniques can succeed where general-purpose SAT-based approaches fall short, all while providing a simple and efficient synthesis engine.

Inspired by a classical synthesis technique rooted in symmetry detection through spectral methods, our approach builds on the seminal work of Edwards et al. [5]. They proposed combining symmetry detection with adding operators to the circuit representation using a synthesis procedure known as *remapping*. While their method was implemented as an interactive program, the authors highlighted the need to identify suitable synthesis options automatically.

Our approach enhances synthesis by analytically deriving equations to estimate the benefits of each step, enabling efficient automation using simple yet effective metrics. The engine supports both manual and automated synthesis, using an analytical model to predict the expected reward based on the number of *don't-cares* generated, as shown in [5], [6].

The representation produced by the proposed synthesis engine are compared to the mainstream logic synthesis techniques [7]. The results show that our engine not only creates near-optimal representations for large Boolean functions, but also identifies provably optimal sub-structures that can be used for devising bitwise manipulation-based algorithms, demonstrating the value of interpretable synthesis solutions.

II. Background

A. Boolean Basics

Let $\mathbb{B}^n$ be a Boolean space. A *literal* is a Boolean variable or its negation, so its positive value identifies a subset of $\mathbb{B}^n$. A *cube* is a literal or an intersection of literals. We represent the operations $\{\cap, \cup, \subset, \subseteq, \triangle\}$ between cubes in logic notation $\{\cdot, +, <, \leq, \oplus\}$. Also, C' indicates cube complementation.

We equivalently refer to a cube as intersections of literals, as the binary labelling of their negation, or as the decimal value of the binary labelling. For example, the two-literal cubes in terms of variables (x_i, x_j) are equivalently $\mathbb{B}^2$, $\{x_j x_i, x_j x_i', x_j x_i', x_j' x_i'\}$, $\{11, 10, 01, 00\}$, or $\{3, 2, 1, 0\}$.

Two cubes C and D are *independent* if they have an empty intersection ($C \cdot D = 0$). A cube C is *contained* in a cube D if $C \leq D$. For instance, $x_i x_j'$ is contained in x_i.

The *minterms* are the 2^n n-literal cubes contained in $\mathbb{B}^n$. The *cofactor* of f with respect to a cube C is the function f_C of the variables not appearing in C and whose minterms are the subsets of minterms of f contained in C. When C is a minterm, f_C is the value of the function. The minterms can be partitioned based on the value of f_M. If $f_M = 0$, M is in the *offset* of f, otherwise M is in the *onset* of f.

An *incompletely specified* function f is a function whose output is not defined (or is a *don't-care*) for some input minterms, which can happen when f is a sub-function of a larger logic block. *Controllability don't-cares* (CDCs) are patterns never appearing at the function inputs. We represent the CDCs with a function named CDC-*mask* $\mu : \mathbb{B}^n \to \mathbb{B}$. Given a minterm M, $\mu_M \in \mathbb{B}$ indicates if M can appear at the input. Given a function f with mask μ, there are $2^{|\text{CDC}|}$ *compatible functions* $\tau : \mathbb{B}^n \to \mathbb{B}$ obtained by assigning 1s and 0s in all possible ways to the *don't cares* of function f.

A *Boolean chain* for n variables $(x_1, ..., x_n)$ is a sequence $(x_{n+1}, ..., x_{n+r})$ where each step combines two previous steps $x_i = x_{l(i)} \circ_i x_{r(i)} \quad n{+}1 \leq i \leq n{+}r$ where $1 \leq l(i) < r(i) < i$ and $\circ_i$ is a binary Boolean operator [3].

A Boolean function is symmetric if the output value depends uniquely on the number of ones appearing at its inputs. These functions are in the form $S_{k_1,k_2,...,k_m}$, and evaluate to 1 when the number of ones at the input is k_1, k_2, ..., or k_m.

B. Two-Variable Symmetries

Classical symmetries correspond to functional equivalences in subspaces defined by two-literal cubes. Cofactor comparisons give information on these functional properties.

The fundamental two-variables symmetries are *non-equivalence symmetry* (NES), *equivalence symmetry* (ES), and *multiform symmetry* (MS). Fig. 1(a-c), show these symmetries on the Karnough map (KM). The explicit functional form is:

1) NES$\{x_i, x_j\}$: $f_{01} = f_{10}$.
2) ES$\{x_i, x_j\}$: $f_{00} = f_{11}$.
3) MS$\{x_i, x_j\}$: $f_{01} = f_{10} \wedge f_{00} = f_{11}$.

The red numbers offer an example in which f is a two-input function of x_i and x_j. In the general case, the equivalence identified by the lines is among cofactors. For instance, Fig. 1(b) means that $f_{00} = f_{11}$, where $f_{00}, f_{11} : \mathbb{B}^{n-2} \to \mathbb{B}$. In the presence of *don't-cares*, they can be allocated to complete a cofactor and satisfy a symmetry. Fig. 1(b) offers an example: assigning the *don't-care* $*$ to 0 results in ES.

Single-variable symmetries (SVS) correspond to cofactor equalities in the space identified by a single variable:

1) $\{\text{SVS}x_i\}x_j : f_{10} = f_{11}$.
2) $\{\text{SVS}x_i\}x'_j : f_{00} = f_{01}$.
3) $\{\text{SVS}x_j\}x_i : f_{11} = f_{01}$.
4) $\{\text{SVS}x_j\}x'_i : f_{10} = f_{00}$.

Fig. 1 (d-e) demonstrates two of these symmetries.

Fig. 1 (f) shows that more SVSs can be present at the same time. They are called *compatible single-variable symmetries* (CSVS). The compatibility check in the presence of *don't-cares* can result in an incorrect result, induced by conflicting assignments of *don't-cares*. To avoid this, it is sufficient to combine the SVS-check with an ES-check. The compatible symmetries and their equivalence checks are:

1) CSVS$\{x'_j, x'_i\}$: $f_{00} = f_{01} = f_{10}$.
2) CSVS$\{x'_j, x_i\}$: $f_{00} = f_{01} = f_{11}$.
3) CSVS$\{x_j, x'_i\}$: $f_{00} = f_{10} = f_{11}$.
4) CSVS$\{x_j, x_i\}$: $f_{01} = f_{10} = f_{11}$.

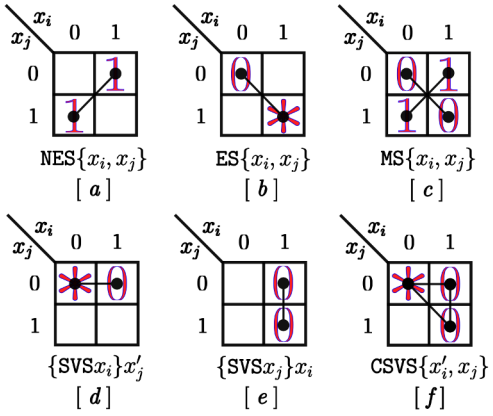

Fig. 1. Schematic representations of the symmetry classes.

C. Synthesis Method Using Symmetries

Edwards et al. [5] devised a synthesis strategy based on *remapping*, i.e., the exploitation of two-variable symmetries to progressively map the problem to a simpler one. The simplicity of the problem corresponds to the fact that each synthesis stage increases the number of CDCs. Hence, at each synthesis stage, the number of exploitable symmetries is either the same or higher. Sec. II-B shows that two-variable symmetries correspond to the equivalence of a function in the subspace identified by two two-literal cubes. Given two variables x_i and x_j, let $\Pi = (C_k)_{k=0}^3$ be any permutation of the two-literal cubes in $\mathbb{B}^2$. If we define the *simple symmetries* (SS) as the group of symmetries including NES, ES, and SVS, we observe

1) SS $\Leftrightarrow \exists\, p \neq q$: $f_{C_p} = f_{C_q}$.
2) CSVS $\Leftrightarrow \exists\, p \neq q \neq r$: $f_{C_p} = f_{C_q} = f_{C_r}$.
3) MS $\Leftrightarrow \exists\, p \neq q \neq r \neq s$: $f_{C_p} = f_{C_q}$ and $f_{C_r} = f_{C_s}$.

All the subscripts in the previous equalities are distinct. We then group the cubes into three sets $\mathcal{S}$, $\mathcal{D}$, and $\mathcal{N}$, defined as:

1) SS : $\mathcal{S} = (C_p)$ $\mathcal{D} = (C_q)$ $\mathcal{N} = (C_r, C_s)$.
2) CSVS : $\mathcal{S} = (C_p, C_q)$ $\mathcal{D} = (C_r, C_r)$ $\mathcal{N} = (C_s)$.
3) MS : $\mathcal{S} = (C_p, C_q)$ $\mathcal{D} = (C_r, C_s)$ $\mathcal{N} = \varnothing$.

From the definitions of SS, CSVS, and MS in terms of C_p, C_q, and C_r, replacing the i-th cube in $\mathcal{S}$ with the corresponding i-th cube in $\mathcal{D}$ at the inputs of f, while preserving the cubes in $\mathcal{D} \cup \mathcal{N}$, does not affect the evaluation of f. This observation is formalized by *remapping*, which defines the following map:

$$\varphi : \mathbb{B}^2 \to \mathbb{B}^2 \qquad \begin{cases} \varphi(C_k) = D_k & C_k \in \mathcal{S}, D_k \in \mathcal{D} \\ \varphi(C_k) = C_k & C_k \in \mathcal{D} \cup \mathcal{N} \end{cases} \tag{1}$$

Explicitly, given two symmetric variables x_i, x_j, the map reads

$$(x_j, x_i) \to (\varphi_j(x_j, x_i), \varphi_i(x_j, x_i)) \begin{cases} 00 \to \varphi(00) \\ 01 \to \varphi(01) \\ 10 \to \varphi(10) \\ 11 \to \varphi(11) \end{cases} \quad (2)$$

Figure 2 shows the map for NES, which is a SS identified by $\mathcal{S} = (10)$, $\mathcal{D} = (01)$, and $\mathcal{N} = (00, 11)$. The map identifies the truth tables of two Boolean operators. Substituting x_j and x_i with the newly defined variables, results in an equivalent synthesis problem named *remapped problem*. The *remapped problem* is simpler in the sense that the minterms contained in the cubes in $\mathcal{S}$ will not appear in the remapped problem, enlarging the set of the CDC_f at the next synthesis stage. As soon as a one-literal cube C is not observable, it can be removed from the inputs list, so that the progressive increase of the CDC set eventually results in obtaining a unique variable, synthesizing the function. Fig. 2 shows the detail of a remapping step in presence of NES. More details for the different symmetries can be found in [5].

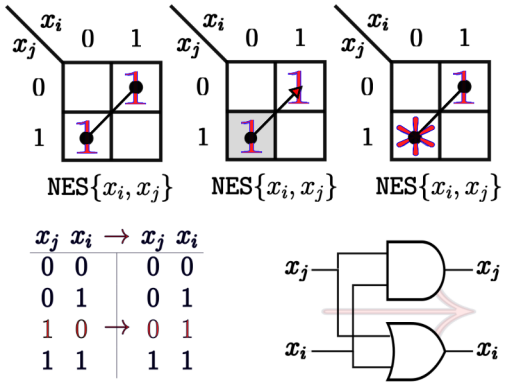

Fig. 2. Remapping induced by NES$\{x_i, x_j\}$.

III. Automated Symmetry-Based Synthesis

This section translates the remapping principle, as described in section II-C, into a set of analytical equations designed to guide the synthesis process. Adding a correct operation to the Boolean chain brings us closer to the final solution, simplifying the sub-problem of completing the remaining entries. To capture this, we define cost functions that assess the number of *don't-cares* generated by adding an operator.

A. Symmetry Detection

Differently from the seminal paper [5], we check the presence of symmetries directly using truth tables. Let f be an incompletely specified function with CDC-mask μ. If A and B are two-literal cubes defining a symmetry via the equivalence $f_A = f_B$, then all the compatible functions τ satisfy

$$f_A = f_B \Leftrightarrow \mu_A \mu_B \tau_A = \mu_A \mu_B \tau_B \quad (3)$$

Indeed, this equation encodes the following cases:

1) if $A \in \text{CDC}_f \vee B \in \text{CDC}_f$ then $f_A = f_B \Leftrightarrow 0 = 0$
2) if $A \in \text{CDC}'_f \wedge B \in \text{CDC}'_f$ then $f_A = f_B \Leftrightarrow \tau_A = \tau_B$

Hence, checking the presence of a symmetry for truth tables corresponds to the bitwise verification of its definition.

B. Remapping Equations

For the sake of readability, we rename the cubes of the partition of $\mathbb{B}^2$ introduced in Sec. II-C: $\Pi = \{A, B, C, D\}$. We consider the general remappings:

1) SS : $A \mapsto C$.
2) CSVS : $A \mapsto C$ and $B \mapsto C$.
3) MS : $A \mapsto C$ and $B \mapsto D$.

The *remapping equations* are the transformations of f from step t to step $t+1$, i.e., the assignment $\tau^{t+1}, \mu^{t+1} \leftarrow \tau^t, \mu^t$.

The remapping equations for the mask are:

$$\mu \xleftarrow{\text{SS}} A'\mu + C\mu_A$$
$$\mu \xleftarrow{\text{CSVS}} A'B'\mu + C(\mu_A + \mu_B)$$
$$\mu \xleftarrow{\text{MS}} A'B'\mu + (C\mu_A + D\mu_B)$$

The first term in the disjunctions removes the minterms contained in the source cubes from the *care set*. The second term considers the possibility that a remapping process restores a minterm from the CDC set. Indeed, cofactoring the mask to a source cube A identifies all cubes K in the remaining variables that, when put in conjunction with A are in the *care set* ($KA \in \text{CDC}'_f$). Hence, after the conjunction with the target cube C, the new minterms generated by the remapping process are in the *care set* ($KC \in \text{CDC}'_f$).

The remapping equations for τ are

$$\tau \xleftarrow{\text{SS}} B'\tau + B(\mu_B\tau + \mu_A\tau_A)$$
$$\tau \xleftarrow{\text{CSVS}} (C' + \mu_C)\tau + C \cdot (\mu_A\tau_A + \mu_B\tau_B)$$
$$\tau \xleftarrow{\text{MS}} (C'D' + C\mu_C + D\mu_D)\tau + (C\mu'_C\mu_A\tau_A + D\mu'_D\mu_B\tau_B)$$

Also in this case there are two terms in the truth table computation. The first contribution demands that the function should be the same for all the minterms that were observable at time t. Meanwhile, the second contribution takes care of the fact that if there is a CDC in the source cubes of the remapping, the use of the degree of freedom of the CDC comes at the cost that we must reallocate the value of τ. Indeed, CDCs can become visible at later stages, and ignoring the second contribution would break the equivalence of the remapped problem with the original one.

C. A Reward Function Based on Don't Cares Prediction

In the seminal paper [5], the authors provide two guiding principles for the designer using the interactive tool. Firstly, remapping should not be performed if the destination set $\mathcal{D}$ contains only *don't-cares*. Secondly *care* minterms should not be remapped into the *don't-care* area to avoid reallocating definite values to present *don't-cares*.

The remapping equations allow us to include the first principle in an automatic method. Indeed, if $\{a_{t,i}\}_{i=1}^m$ is the set of available synthesis actions at time t, we can compute the

remapped mask $\mu[a_{t,i}] \xleftarrow{a_{t,i}} \mu$. Next, by defining the *reward* function as the number of CDCs, we can select the action as

$$a^* = \arg\max_a |\texttt{CDC}[a]| = \arg\max_a |\mu[a]|_0 \tag{4}$$

For what concerns the second principle, the remapping equations for τ allow us to perform reallocation when this is advantageous. Our experiments show that CDC-maximization is a powerful guiding principle for symmetric functions, but greedily maximizing the reward function does not always yield the best result (See Section. V-B). We address this limitation by defining an engine that explore multiple solutions in the presence of ties of remapping candidates, and providing more heuristic details that proved effective in guiding synthesis.

D. Remapping-Based Solver

Algorithm 1 describes the structure of our solver. The method takes a Boolean function as input and returns a corresponding Boolean chain, referred to as CHAIN. Depending on the binary Boolean operators used, CHAIN corresponds to one of two widely adopted network types in logic synthesis: if $\circ_i \in \{\wedge, <, >\}$, then CHAIN is an and-inverter graphs (AIGs) [8], [9], if $\circ_i \in \{\wedge, <, >, \oplus\}$, then CHAIN is a xor-and-inverter graph (XAIGs) [10].

The algorithm stores the best result after applying synthesis a number of times. A single synthesis step takes a set of variables named *cut*, initialized to the input variables, and iteratively remaps the cut variables into new variables, while updating the functional information using the remapping equations. The process terminates if the cut contains a single node.

First, we initialize $\texttt{state}_t$, which is the encoding of the partial solution. The $\texttt{state}_t$ object represents the partial *Boolean chain*, not yet synthesizing the function, and characterized with the following attributes: 1) The cut of the last synthesis stage; 2) the target function (τ_t, μ_t); 3) the functions of the cut variables; 4) the CDCs of the cut.

Next, we synthesize the network a few gates at a time. At each step, the $\texttt{analyzer}(\cdot)$ function identifies a set of candidate moves from the current state of the solution. The analyzer function evaluates the symmetries in the current state, and enumerate all possible remapping-based actions using Equation 3. For each candidate action, the remapping equations for the mask allow us to compute the number of CDCs induced by each action.

After selecting an action using an iteration-dependent $\texttt{policy}(\cdot, \texttt{it})$, $\texttt{move}(\cdot)$ updates the partial solution. When $\texttt{state}_t$ corresponds to a circuit satisfying the specifications defined by f, we synthesize the desired network representation. The method returns the smallest size representation identified in the synthesis process.

If only one iteration of the engine is performed, we prioritize symmetries acting on the same groups of variables. Intuitively, we do so to eliminate variables from the representation as soon as possible. This policy detail also allows us to identify recursive sub-structures when running the engine in the non-automatic mode. When considering multiple iterations, say N, we relax this filter to explore more synthesis solutions, and choose uniformly at random among actions with the same reward. Moreover, when multiple logic blocks yield the same *don't care* gain, we prioritize lower numbers of Boolean operators. The difference arises when synthesizing AIGs over XAIGs since the number of Boolean operators to represent an XOR (MS-remapping) in an AIG is 3.

Algorithm 1 CHAIN chain ← SOLVER<CHAIN>(f)

```
|chain_best| ← ∞
while ( it++<number of iterations ) do
  state_t ← initialize(f)
  while ( f not satisfied ) do
    actions ← analyzer(state_t)
    action_t ← policy(actions, it)
    state_{t+1} ← move(state_t, action_t)
  chain ← synthesize<CHAIN>(state_t)
  if |chain| < |chain_best| then
    chain_best ← chain
return chain_best
```

IV. A Boolean Evaluation-Based Algorithm

This section shows that synthesizing compact representations of Boolean functions allows for designing efficient algorithms based on Boolean evaluation, providing a concrete example of the importance of symmetric functions.

A. Recursive Sub-Structure of the One-Hot Encoding

Table II shows the number of binary operators needed to represent threshold functions and k-hot encoding functions. Interestingly, it is possible to see that the 1-hot encoding chain for n inputs differs from the one for $n+1$ inputs by 3 gates. This section shows that this regularity is due to the recursive sub-structure of this function, thanks to which it is possible to write the Boolean chain for arbitrary input size.

Lemma IV.1 (One-hot encoding)**.** *The combinational complexity of the n-inputs one-hot encoding is $\mathcal{C}(S_1^n) = 3n - 2$.*

Proof. For $n = 2$, $S_1^2 = x_1 \oplus x_0$. Hence, $\mathcal{C}(S_1) = 1 = 3 - 1$.
Let us define the map $\varphi : \mathbb{B}^n \mapsto \mathbb{B}^{n-1}$:

$$\varphi_i(\{x_i\}_{i=1}^n) = \begin{cases} x_i & \text{if } i < n-2 \\ x_{n-1} \vee x_n & \text{if } i = n-2 \\ x_{n-2} \vee (x_{n-1} \wedge x_n) & \text{if } i = n-1 \end{cases} \tag{5}$$

We classify the minterms based on the number of ones in (x_n, x_{n-1}, x_{n-2}). The minterm with k ones is M_k. Then,

1) $S_1^n(M_0) = S_1^{n-1}(\varphi(M_0)) = S_1^{n-3}(\{x_i\}_{i=1}^{n-3})$
2) $S_1^n(M_1) = S_1^{n-1}(\varphi(M_1)) = \bigwedge_{i=1}^{n-3} x_i'$
3) $S_1^n(M_2) = S_1^{n-1}(\varphi(M_2)) = 0$
4) $S_1^n(M_3) = S_1^{n-1}(\varphi(M_3)) = 0$

Which yields the recursive formula $S_1^n(x) = S_1^{n-1}(\varphi(x))$ with termination $S_1^2(x) = x_1 \oplus x_0$.

Since the combinational complexity of the map is $\mathcal{C}(\varphi) = 3$, $n-1$ recursive steps are needed to reach the termination condition, and the combinational complexity of the terminating structure is $\mathcal{C}(S_1^2) = 1$, the combinational complexity is $C(S_1^n) = 3(n-1) + 1 = 3n - 2$ □

B. A Linear-Time Algorithm to Find Essential Sets

The set covering problem is a significant NP-hard problem in combinatorial optimization [11]. Given a collection of elements U, the goal is to find the minimum number of sets that cover these elements. The Boolean evaluation of the one-hot-encoding function yields an efficient bit-manipulation algorithm to identify sets that must be included in the solution.

Proposition IV.1 (Essential columns)**.** *Let $\{S_i\}_{i=1}^{|\mathcal{S}|}$ be the number of sets in a set covering problem $\mathcal{S} = \{S_i\}_{i=1}^{|\mathcal{S}|}$. The detection of the* essential sets *requires $4|\mathcal{S}| - 2$ set operations.*

Proof. We represent U and each set S_i as a truth-table in the space $\mathbb{B}^n$, s.t., $2^n \geq |\mathcal{U}|$. Then, the algorithm goes as follows

1) Using Lemma IV.1, $B = S_1^{|\mathcal{S}|}(\{S_i \cdot U\}_{i=1}^{|\mathcal{S}|}) \in \mathbb{B}^{r\times 1}$ is computed in $3|S| - 2$ operations. $B_i = 1$ iff the element i present in f is covered by a single column of A.
2) Foreach set S_i, if $|A^{S_i} \cdot B|_1 > 0$ the set is essential. This step requires $|\mathcal{S}|$ steps in the worst case.

The overall computational cost is $3|\mathcal{S}|-2+|\mathcal{S}| = 4|\mathcal{S}|-2$ □

V. Experiments

In this section, we evaluate the performances of our solver on symmetric functions. Section V-A compares the synthesis quality obtained by the engine with a state-of-the-art technique. Section V-B evaluates the assumptions of our heuristic. Finally, Section V-C investigates the quality of the solvers on functions with known combinational complexity. The experiments were conducted on a machine with an `i7-1365U` CPU.

A. One-Shot Synthesis and Comparison with State-of-the-Art

This experiment has two goals: to verify if high-effort optimization can improve the results produced by our heuristics, and to compare the results against those produced by state-of-the-art synthesis and optimization.

Table I shows the experimental results. We focus on a subset of the *threshold* functions named *majority* functions, evaluating to 1 when the majority of the inputs is 1. We vary the input size from 3 to 20, and report the quality of the results when running the version of the heuristic introducing most a-priori information in the definition of the reward function.

As a first test, we perform aggressive optimization on top of the result obtained with our solver. We use the ABC command `deepsyn`. `Deepsyn` contains a random initialization procedure on which the quality of results depends. We consider 20 random initializations and halt the optimization after 500 iterations without any improvement (`&get; &deepsyn -I 20 -J 500;&put;`). The entries of type $X \mapsto Y$ show that an AIG with X gates can be optimized to an AIG of size Y. Column `T[s]` reports the synthesis time of our procedure. With up to 17 inputs, we can synthesize functions in less than a second. High-effort optimization fails in finding optimizations in most of the representations obtained by our engine, showing the stability of the minimum found by our solver.

The absence of optimization does not give a guarantee on the globality of the minimum. To obtain a strong baseline, we synthesize the functions as *functionally reduced* AIGs (FRAIG), that are compact AIGs in which no two nodes have the same functionality [9]. Next, we perform aggressive optimization. Column `|FRAIG|` reports the number of AND gates in the FRAIG generated from the truth table. Next, we repeat `deepsyn`-based optimization with high-effort hyperparameter tuning to obtain the most compact representation possible, whose size is indicated as `|FRAIG*|`. The real-world time to obtain most of the results in Table I is of the order of minutes. However, we report an underestimation of the time invested by removing the time needed for the hyperparameter optimization and the time invested by `deepsyn` in performing unsuccessful optimizations. Despite the aggressive optimization and the underestimation of the state-of-the-art synthesis time, our method finds the same solution or a better one in most cases, and our engine runs within milliseconds for all the functions that we can represent as truth tables in ABC [7] ($n \leq 16$).

TABLE I
Comparison of our solver and FRAIG-based synthesis.

	HYPERPARAMETER TUNING			OUR ENGINE$_{\times 1}$	
n	|FRAIG|	|FRAIG*|	T$_\infty$[s]	|AIG|	T[s]
3	6	**4**	0.02	**4**	**0.00**
4	13	**7**	0.02	**7**	**0.00**
5	23	**10**	0.02	**10**	**0.00**
6	36	**15**	0.05	**15**	**0.00**
7	68	**20**	1.94	**20**	**0.00**
8	99	**25**	2.59	**25**	**0.01**
9	142	**30**	14.52	**30**	**0.01**
10	207	40	23.61	**37**	**0.02**
11	301	49	13.47	**44**	**0.04**
12	452	**49**	29.18	**49**	**0.05**
13	637	64	33.09	**58**	**0.07**
14	985	**62**	31.54	$67 \mapsto 66$	**0.10**
15	1357	**71**	86.46	72	**0.18**
16	2155	105	120.76	**79**	**0.45**
17	–	–	–	**92**	**1.27**
18	–	–	–	**101**	**2.62**
19	–	–	–	**116**	**12.86**
20	–	–	–	**121**	**32.39**

B. Evaluating the Reward Function

The one-shot version of the solver uses several assumptions: it considers only actions that maximize the number of remapped *don't cares*, prioritizes those introducing the fewest Boolean operators into the representation, and favors exploiting symmetries within the same subgroup of variables across iterations. The last synthesis policy comes from the fact that actions with the same reward are equally likely to map the problem to a simpler one. Likewise, selecting actions involving variables manipulated in previous synthesis steps is likely to induce the removal of variables, simplifying the problem.

Table II shows the results for two classes of functions:

- The *threshold* functions $S_{\geq k}$ evaluate to 1 when there are at least k ones at the input.
- The *k-hot-encoding* functions S_k evaluate to 1 when there are k ones at the input.

The functions not represented in the table relate to one entry by duality. The vertical line identifies the *majority function*.

We consider 33 iterations of the solver and indicate the quality of the results as ${}_\delta X^\Delta$, where:

1) X is the minimal number of Boolean operators observed during synthesis.

TABLE II
XAIGs FOR THRESHOLD FUNCTIONS AND k-HOT ENCODING FUNCTIONS.

THRESHOLD FUNCTIONS

n	$S_{\geq 1}$	$S_{\geq 2}$	$S_{\geq 3}$	$S_{\geq 4}$	$S_{\geq 5}$	$S_{\geq 6}$	$S_{\geq 7}$	$S_{\geq 8}$	$S_{\geq 9}$	$S_{\geq 10}$
2	${}_0 1^0$	${}_0 1^0$								
3	${}_0 2^0$	${}_0 4^0$								
4	${}_0 3^0$	${}_0 7^0$	${}_0 7^0$							
5	${}_0 4^0$	${}_0 10^0$	${}_0 10^0$							
6	${}_0 5^0$	${}_0 13^0$	${}_0 15^0$	${}_0 15^0$						
7	${}_0 6^0$	${}_0 16^0$	${}_0 18^0$	${}_0 20^0$						
8	${}_0 7^0$	${}_0 19^0$	${}_0 21^0$	${}_0 25^0$	${}_0 25^0$					
9	${}_0 8^0$	${}_0 22^0$	${}_0 24^0$	${}_0 30^0$	${}_0 30^2$					
10	${}_0 9^0$	${}_0 25^0$	${}_0 29^0$	${}_0 35^0$	${}_0 37^2$	${}_0 37^2$				
11	${}_0 10^0$	${}_0 28^0$	${}_0 32^0$	${}_0 40^0$	${}_0 42^4$	${}_0 44^4$				
12	${}_0 11^0$	${}_0 31^0$	${}_0 35^0$	${}_0 45^0$	${}_2 49^4$	${}_0 49^6$	${}_0 49^6$			
13	${}_0 12^0$	${}_0 34^0$	${}_0 38^0$	${}_0 50^0$	${}_2 52^6$	${}_0 56^6$	${}_2 58^4$			
14	${}_0 13^0$	${}_0 37^0$	${}_0 41^0$	${}_0 55^0$	${}_2 61^4$	${}_0 63^6$	${}_2 63^8$	${}_2 65^8$		
15	${}_0 14^0$	${}_0 40^0$	${}_0 44^2$	${}_0 60^0$	66^4	${}_0 70^6$	${}_0 72^6$	${}_0 72^{10}$		
16	${}_0 15^0$	${}_0 43^0$	${}_0 47^0$	${}_0 65^0$	${}_2 73^4$	${}_0 77^8$	${}_0 79^{10}$	${}_0 79^{16}$	${}_0 79^{18}$	
17	${}_0 16^0$	${}_0 46^0$	${}_0 50^0$	${}_0 70^0$	${}_0 76^{10}$	${}_2 82^8$	${}_2 88^8$	${}_0 90^{10}$	${}_0 92^{12}$	
18	${}_0 17^0$	${}_0 49^0$	${}_0 55^0$	${}_0 75^0$	${}_0 87^4$	${}_0 89^{10}$	${}_0 93^{12}$	${}_0 99^{10}$	${}_0 99^{12}$	${}_0 101^{10}$

k-HOT ENCODING FUNCTIONS

n	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_9	S_{10}
2	${}_0 1^0$	${}_0 1^0$								
3	${}_0 4^0$	${}_0 4^0$								
4	${}_0 7^0$	${}_0 7^0$	${}_0 7^0$							
5	${}_0 10^0$	${}_0 12^0$	${}_0 12^0$							
6	${}_0 13^0$	${}_0 15^0$	${}_0 17^0$	${}_0 15^0$						
7	${}_2 16^2$	${}_2 18^2$	${}_0 22^0$	${}_0 22^0$						
8	${}_0 19^0$	${}_0 21^0$	${}_0 27^0$	${}_0 27^4$	${}_0 27^0$					
9	${}_0 22^0$	${}_0 26^0$	${}_0 32^0$	${}_0 32^4$	${}_0 32^4$					
10	${}_0 25^0$	${}_0 29^0$	${}_0 37^0$	${}_2 39^4$	${}_2 39^6$	${}_2 39^4$				
11	${}_0 28^0$	${}_0 32^0$	${}_0 42^0$	${}_0 44^4$	${}_2 46^6$	${}_0 46^6$				
12	${}_0 31^0$	${}_0 35^0$	${}_0 47^0$	${}_2 51^6$	${}_2 53^6$	${}_0 53^8$	${}_0 53^6$			
13	${}_0 34^0$	${}_2 38^2$	${}_0 52^0$	${}_0 56^6$	${}_2 60^8$	${}_0 62^8$	${}_0 62^8$			
14	${}_0 37^0$	${}_0 41^0$	${}_0 57^0$	${}_0 65^4$	${}_0 67^8$	${}_0 69^8$	${}_0 71^8$	${}_4 67^{10}$		
15	${}_0 40^0$	${}_2 44^2$	${}_0 62^0$	${}_0 68^8$	${}_4 72^8$	${}_2 78^8$	${}_4 76^{10}$	${}_0 78^{10}$		
16	${}_0 43^0$	${}_0 47^0$	${}_0 67^0$	${}_0 77^6$	${}_4 81^8$	${}_2 83^{10}$	${}_0 85^{14}$	${}_2 87^{12}$	${}_0 87^{10}$	
17	${}_0 46^0$	${}_0 52^0$	${}_0 72^0$	${}_0 80^8$	${}_2 86^8$	${}_2 94^{10}$	${}_0 98^{10}$	${}_0 98^{12}$	${}_0 98^{16}$	
18	${}_0 49^0$	${}_0 55^0$	${}_0 77^0$	${}_0 89^6$	${}_0 91^{14}$	${}_0 97^{12}$	${}_4 103^{12}$	${}_0 105^{16}$	${}_0 109^{14}$	${}_0 105^{18}$

2) δ is the difference between the number of Boolean operators obtained by the one-shot version and X.
3) Δ is the difference between the maximal number of Boolean operators observed during synthesis and X.

We highlight in red (${}_\delta X^\Delta$) the cases in which the one-shot version yields an empirically provable sub-optimal result. This experiment empirically validates the a-priori assumptions of the one-shot approach. Nevertheless, in some cases, breaking ties at random can identify more compact representations. This remark is interesting in light of the analysis done in Section V-A: multiple runs of fast synthesis with stochastic selections of moves with high expected rewards can find solutions that are hardly reachable with traditional engines.

C. Comparison with Exact Synthesis

In this experiment, we compare the quality of results for four- and five-input symmetric functions with known combinational complexity [3]. Table III shows the results. The notation $X \mapsto Y$ indicates that applying an XAIG optimization algorithm [12] on top of the synthesis result X allows us to further reduce the gate count to Y. We also ran the same experiment with 100 iterations, but there was no considerable variation in the results, confirming the effectiveness of our single-pass heuristic. This experiment shows that the engine can find many close to minimum circuits in at most $10ms$.

VI. CONCLUSIONS

We propose a synthesis engine based on Edward's theory of symmetry-based remapping. We empirically investigated the effectiveness of this method by synthesizing two-input gate circuits for several classes of symmetric functions. We showcase the effectiveness of the proposed synthesis engine by showing that it outperforms state-of-the-art synthesis and optimization. While the engine fails to find the known minimum circuits for some functions, the regularity of the resulting circuits allow for identifying a recursive sub-structure of a Boolean function, which can be used to create bitwise manipulation algorithms.

TABLE III
COMPARISON OF EXACT XAIGs SYNTHESIS WITH OUR ENGINE.

$n=4$			$n=5$		
f	$\mathcal{C}(f)$ [3]	|XAIG|	f	$\mathcal{C}(f)$ [3]	|XAIG|
S_3	7	7	S_4	10	10
S_4	3	3	$S_{4,5}$	10	10
$S_{3,4}$	7	7	S_3	9	12
S_2	6	7	$S_{3,5}$	10	10
$S_{2,4}$	6	7	$S_{3,4}$	10	13
$S_{2,3}$	6	$9 \mapsto 8$	$S_{3,4,5}$	9	10
$S_{2,3,4}$	7	7	$S_{2,5}$	10	14
S_1	7	7	$S_{2,4}$	8	10
$S_{1,4}$	7	$9 \mapsto 8$	$S_{2,4,5}$	9	12
$S_{1,3}$	3	3	$S_{2,3,5}$	10	15
$S_{1,3,4}$	6	7	$S_{2,3}$	8	15
$S_{1,2}$	6	$9 \mapsto 8$	$S_{2,3,4}$	10	$13 \mapsto 12$
$S_{1,2,4}$	7	$9 \mapsto 8$	$S_{1,5}$	9	$13 \mapsto 12$
$S_{1,2,3}$	5	7	$S_{1,4}$	9	15
$S_{1,2,3,4}$	3	3	$S_{1,3,4}$	11	13

REFERENCES

[1] M. Soeken, W. Haaswijk, E. Testa, A. Mishchenko, L. G. Amarù, R. K. Brayton, and G. De Micheli, "Practical exact synthesis," in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2018, pp. 309–314.

[2] W. Haaswijk, M. Soeken, A. Mishchenko, and G. De Micheli, "Sat-based exact synthesis: Encodings, topology families, and parallelism," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 4, pp. 871–884, 2019.

[3] D. E. Knuth, *The art of computer programming, volume 4A: combinatorial algorithms, part 1*. Pearson Education India, 2011.

[4] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1," *arXiv preprint arXiv:1602.02830*, 2016.

[5] C. R. Edwards and S. L. Hurst, "A digital synthesis procedure under function symmetries and mapping methods," *IEEE Transactions on Computers*, vol. 27, no. 11, pp. 985–997, 1978.

[6] C. Scholl, "Multi-output functional decomposition with exploitation of don't cares," in *Proceedings Design, Automation and Test in Europe*. IEEE, 1998, pp. 743–748.

[7] R. Brayton and A. Mishchenko, "Abc: An academic industrial-strength verification tool," in *Computer Aided Verification: 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings 22*. Springer, 2010, pp. 24–40.

[8] A. Kuehlmann and F. Krohm, "Equivalence checking using cuts and heaps," in *Proceedings of the 34th annual Design Automation Conference*, 1997, pp. 263–268.

[9] A. Mishchenko, S. Chatterjee, R. Jiang, and R. K. Brayton, "Fraigs: A unifying representation for logic synthesis and verification," ERL Technical Report, Tech. Rep., 2005.

[10] G. Meuli, M. Soeken, and G. De Micheli, "Xor-and-inverter graphs for quantum compilation," *npj Quantum Information*, vol. 8, no. 1, p. 7, 2022.

[11] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT press, 2022.

[12] S.-Y. Lee, H. Riener, A. Mishchenko, R. K. Brayton, and G. De Micheli, "A simulation-guided paradigm for logic synthesis and verification," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 8, pp. 2573–2586, 2021.