SAS – A Framework for Symmetry-based Approximate Synthesis

Niklas Jungnitz
niklas.jungnitz@fau.de
Friedrich-Alexander-Universität Erlangen-Nürnberg
Erlangen, Germany

Oliver Keszocze
oliver.keszoecze@fau.de
Friedrich-Alexander-Universität Erlangen-Nürnberg
Erlangen, Germany

ABSTRACT

Approximate Computing is a design paradigm that trades off computational accuracy for gains in non-functional aspects such as reduced area, increased computation speed, or power reduction. The latter is of special interest in the field of Internet of Things. In this paper we present SAS, a framework for symmetry-based approximate logic synthesis. Given a Boolean multi-output function, SAS approximates it by (partially) replacing its output functions by symmetric functions with minimal Hamming distance. The framework is capable of restricting the introduced error with respect to a parameterized error metric that covers many real-word use-cases.

Experimental results on common benchmark sets as well as large bit width arithmetic Boolean functions confirm the effectiveness of the proposed framework. SAS is capable of synthesizing Boolean functions with size reductions of up to ≈ 45% while, at the same time, respecting the specified threshold on the error metric. The framework is publicly available as open-source software on GitHub.

KEYWORDS

Approximate Computing, BDD, AIG, Optimization, Logic Synthesis

ACM Reference Format:
Niklas Jungnitz and Oliver Keszocze. 2024. SAS – A Framework for Symmetry-based Approximate Synthesis. In *61st ACM/IEEE Design Automation Conference (DAC '24), June 23–27, 2024, San Francisco, CA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3649329.3658495

ACKNOWLEDGMENTS

The paper has been partially funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – 450987171.

1 INTRODUCTION

Many applications in the domain of digital signal processing, such as image processing, do not require computations to be exact (see, e.g. [1]) as the interpretation of the computed results is limited by non-perfect human perception. The idea behind *Approximate Computing* (AC) is to exploit this fact by trading off computational accuracy for gains in non-functional aspects such as reduced area, increased computation speed, or power reduction. For a good introduction to and overview over Approximate Computing, the interested reader is referred to [2].

DAC '24, June 23–27, 2024, San Francisco, CA, USA

ACM ISBN 979-8-4007-0601-1/24/06
https://doi.org/10.1145/3649329.3658495

The AC synthesis task investigated in this paper is as follows: Given a Boolean function f determine a function $\hat{f}$ that approximates the original function (i.e., $f \approx \hat{f}$) and is smaller than the original function. Within the context of this paper, the quality of the approximation is measured computing the error metric between f and $\hat{f}$ and "smaller" is measured with respect to the size of the function representation of f and $\hat{f}$.

Many approaches for approximate synthesis have been published in the past. The authors of [3] use cubes to represent Boolean functions. This function representation allows for a rather scalable synthesis, including multi-output functions. It is, unfortunately, not applicable for arithmetic functions. In [4], the authors use a combination of Cartesian Genetic Programming and Boolean Satisfiability to synthesize approximate circuits.

A function representation that is used very commonly in the related work are *Binary Decision Diagrams* (BDDs) and *And-Inverter Graphs* (AIGs). In [5], an evolutionary algorithm working directly on the BDD representation of designs is proposed. It automatically adapts its preference for either the BDD size or the error metric value in order to ensure that no good solution is lost due to optimizing too much for a small error. In [6], a greedy bucket-based algorithm for approximating designs that employs rules for successively applying approximation operations in a greedy fashion is presented. The authors of [7] (partially) symmetrize a Boolean function and then represent it as a BDD.

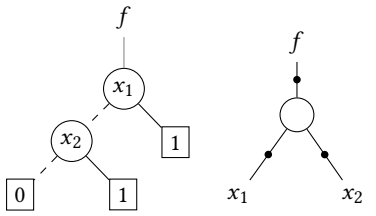
Symmetric functions are desirable as for an n-input function, the size of a corresponding BDD is bounded by n^2 for any variable-ordering [8]. The authors of [7] also sketch an idea how to synthesize a circuit of linear size from a symmetric function. They, however, only support the *error rate* error metric; a metric not particularly suitable for arithmetic functions. This is a serious drawback as this class of functions is of main interest in AC (this is true for other approaches as well, see, e.g., [3]).

In this paper, we present SAS, the Symmetry-based Approximate Synthesis framework, which is based on findings presented in [7]. The main idea behind SAS is, given a Boolean function f, to determine a symmetric function $\tilde{f}$ that is close to f with respect to the Hamming distance between the functions. We introduce a parameterized error metric that exploits the already computed Hamming distance values in its computation. This metric is capable of correctly evaluating arithmetic functions. The synthesis task then is to determine the best set of symmetric output functions so that the overall function size is minimized while, at the same time, the introduced error remains within a predefined limit. For this, we introduce a relaxed optimization problem and prove its equivalence to the Knapsack problem.

The contributions of this paper as summarized as follows:

- An approximate synthesis framework natively supporting BDDs as input and BDDs and AIGs as output
- Usage of a generic, parameterized error metric allowing to cover arithmetic functions

x_1	x_0	f	g
0	0	0	1
0	1	1	$*$
1	0	1	0
1	1	1	0



a Truth Table **b BDD** **c AIG**

Figure 1: Function representations for Boolean functions $f,g:\mathbb{B}^2\to\mathbb{B}$: a truth table for both functions (b–c) BDD and AIG for f.

- A thorough evaluation of the approach including practical benchmarks for arithmetic functions (ADD, MULT, MAC) as well as maximally asymmetric functions as the most difficult class of functions to approximate via symmetrization
- Availability of the framework on GitHub[1]

2 PRELIMINARIES

The set of Boolean values is denoted $\mathbb{B}:=\{0,1\}$ and the extended set of Boolean values containing *don't care* values $*$ is denoted $\mathbb{B}_*:=\{*,0,1\}$.

In this paper we consider the synthesis of fully/partially specified functions of type $f:\mathbb{B}^n\to\mathbb{B}^m/\mathbb{B}_*^m$. The m output functions (also called *components*) are denoted f_i for $1\le i\le m$.

In this work we assume familiarity with Binary Decision Diagrams (BDDs) and And-Inverter-Graphs (AIGs); see Figure 1 for an overview of the function representations used in this paper. By $\#BDD(f)/\#AIG(f)$ we denote the size of the BDD/AIG for the function f.

The *Hamming weight* of a vector $v\in\mathbb{B}^n$ $\mathrm{H}(v)$ is defined as the number of bits that are set to 1, i.e.,

$$\mathrm{H}(v):=\#\{i\mid v_i=1\}.$$

The *Hamming distance* between two vectors $u,v\in\mathbb{B}_*^n$ is defined as

$$\Delta(u,v):=\sum_{i=1}^{n}\begin{cases}0 & u_i=*\ \vee\ v_i=*\\ u_i\oplus v_i & \text{otherwise}\end{cases}.$$

This distance is naturally extended to measure the distance between Boolean functions $f,g:\mathbb{B}^n\to\mathbb{B}_*^m$ as follows

$$\Delta(f,g):=\sum_{x\in\mathbb{B}^n}\Delta(f(x),g(x)).$$

Example 2.1. Consider the two functions f and g in Figure 1a. Their Hamming distance is

$$\begin{aligned}\Delta(f,g)&=\Delta\big(f(0,0),g(0,0)\big)+\ldots+\Delta\big(f(1,1),g(1,1)\big)\\&=\Delta\big(0,1\big)+\Delta\big(1,*\big)+\Delta\big(1,0\big)+\Delta\big(1,0\big)=1+0+1+1=3.\end{aligned}$$

A Boolean function $f:\mathbb{B}^n\to\mathbb{B}$ is *symmetric* if its output value depends on the Hamming weight of the input only. The function f can then be represented by its *value vector* $V_f=\langle v_0,v_1,\ldots,v_n\rangle\in\mathbb{B}^{n+1}$ with v_w being the function value of f for an input with Hamming weight w. This idea naturally extends to multi-output functions.

Example 2.2. Consider again the function f in Table a. The function is symmetric and its value vector is $V_f=\langle 0,1,1\rangle$.

[1] https://github.com/keszocze/sas/

3 ERROR METRICS

In order to assess the quality of the function $\hat{f}$ approximating the function f, various different *error metrics* can be used. The choice of the error metric depends on the aspect of the approximated function that is of interest which, in turn, often depends on the application domain. The interested reader is referred to [9] for an overview of many commonly used error metrics.

In this paper, we start by investigating the *error rate* and the *average case error* which are defined as

$$\mathrm{er}(f,\hat{f}):=\frac{1}{m\cdot 2^n}\cdot\Delta(f,\hat{f}) \tag{1}$$

$$\mathrm{ace}(f,\hat{f}):=\frac{1}{2^n}\sum_{x\in\mathbb{B}^n}|f(x)-\hat{f}(x)|, \tag{2}$$

where in (2) the output of the (multi-output) function f is interpreted as a number.

The error rate is applicable for any Boolean function but does not consider the semantics of the function. The average case error can be used to evaluate approximations of functions where the result is a number, e.g., arithmetic operations such as addition or multiplication. This class of functions is of particular interest in the domain of AC.

We further define the *weighted average error* with weight vector $\alpha\in\mathbb{R}_+^m$ to be

$$\mathrm{wae}_\alpha(f,\hat{f}):=\frac{1}{2^n}\sum_{i=1}^{m}\alpha_i\cdot\Delta(f_i,\hat{f}_i). \tag{3}$$

In the rest of the paper, we will only consider wae_α as a metric as

- with the choice of $\alpha=\langle 1/m,\ldots,1/m\rangle$ we have $\mathrm{wae}_\alpha=\mathrm{er}$.
- with the choice of $\alpha=\langle 2^{m-1},\ldots,2^1,1^0\rangle$ it serves as an upper bound to ace, i.e., $\mathrm{wae}_\alpha(f,g)\ge\mathrm{ace}(f,g)$.
- it easily captures many further use cases.

To allow for a fair comparison between error rates of functions with different numbers of output bits (e.g., to investigate the approximation behavior of adders of different bit widths), we introduce the *normalized weighted average error* as

$$\mathrm{nwae}_\alpha(f,\hat{f}):=\frac{1}{2^m-1}\cdot\mathrm{wae}_\alpha(f,\hat{f}_i).$$

4 SYMMETRY-BASED APPROXIMATE SYNTHESIS (SAS)

This section briefly reviews the symmetry-based approximate synthesis approach as introduced in [7] before generalizing the approach to multiple error metrics and the use of BDDs and AIGs.

4.1 Overview

The general idea of the approach is, given a function $f:\mathbb{B}^n\to\mathbb{B}_*$, to determine a symmetric function $\tilde{f}:\mathbb{B}^n\to\mathbb{B}$ with minimal Hamming distance $\Delta(f,\tilde{f})$.

The synthesis process consists of three steps:

(1) For a function $f:\mathbb{B}^n\to\mathbb{B}_*$, for each $0\le w\le n$ determine how many inputs $x\in\mathbb{B}^n$ with $\mathrm{H}(x)=w$ there are for the output values 1 and $*$. This information is stored in the vectors T (*true*) and D (*don't care*), respectively
(2) Use T and D to construct the value vector $V_{\tilde{f}}$ of the symmetric function $\tilde{f}:\mathbb{B}^n\to\mathbb{B}$ with minimal Hamming distance to f.
(3) Realize the function in a desired representation (e.g., as a BDD or a circuit)

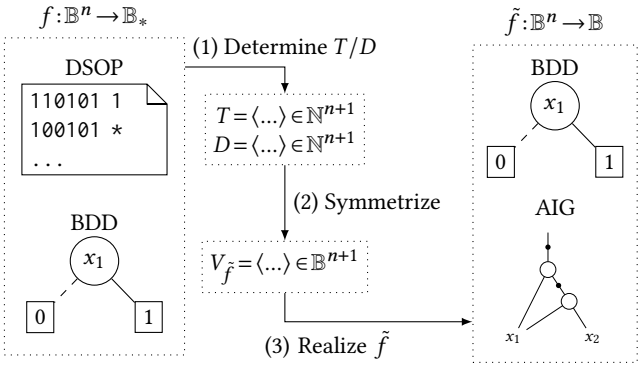


Figure 2: Overview of the symmetrization/approximate synthesis process.

The realized function $\tilde{f}$ then serves as the approximation $\hat{f}$ (i.e., $\hat{f}=\tilde{f}$). The three steps are visualized in Figure 2. Note that the authors of [7] perform (1) and (2) in a single step.

This approach is naturally extended to functions $f:\mathbb{B}^n \to \mathbb{B}_*^m$ by separately symmetrizing the m components f_i, again yielding a symmetric function [7]. The approximation of f is then given by $\hat{f}=(\tilde{f_1},...,\tilde{f_n})$. Hence, it is sufficient to discuss the symmetrization of single output functions.

4.2 Determining T and D

We define the vector $C_{\mathrm{H}}(f,v)\in\mathbb{N}^{n+1}$ as $C_{\mathrm{H}}(f,v)=\langle c(0,v),...,c(n,v)\rangle$ with $c(w,v)=\#\{x\in\mathbb{B}^n \mid f(x)=v\wedge \mathrm{H}(x)=w\}$. This allows to compute the vectors T and D as $T:=C_{\mathrm{H}}(f,1)$ and $D:=C_{\mathrm{H}}(f,*)$.

It is sufficient to develop an algorithm for computing $C_{\mathrm{H}}(f,1)$ for fully specified functions as the computation of $C_{\mathrm{H}}(f,1)$ and $C_{\mathrm{H}}(f,*)$ can then be carried out by computing $C_{\mathrm{H}}(f,1)$ and $C_{\mathrm{H}}(f^*,1)$ with $f^*(x):=(f(x)\Leftrightarrow *)$ instead. Hence, we omit the function value from the vector C_{H} and write only $C_{\mathrm{H}}(f)$ in the rest of this paper.

In [7], an algorithm working on *Disjoint Sum of Products* (DSOP) function representations is presented. This representation is not necessarily the best starting point for the approximate synthesis as a designer might have used an algorithm to produce the to be approximated function f that outputs *Binary Decision Diagrams* (BDDs) instead. In [7], the authors propose to use the technique of [10] to convert a BDD into a DSOP form. Consequently, we present an algorithm to compute $C_{\mathrm{H}}(f)$ directly on BDDs based on the following theorem.

Theorem 4.1. *Let $f:\mathbb{B}^n\to\mathbb{B}$ a function whose first $d\in\mathbb{N}$ variables have no influence on the function's value, i.e., there is a function $g:\mathbb{B}^{n-d}\to\mathbb{B}$ such that*

$$g(x_{d+1},x_{d+2},...,x_n)=f(x_1,x_2,...,x_n), \tag{4}$$

then $C_{\mathrm{H}}(f)$ is given by the sum

$$\sum_{i=0}^{d}\binom{d}{i}\cdot\left(\langle 0\rangle^{d-i}\circ C_{\mathrm{H}}(g)\circ\langle 0\rangle^{i}\right), \tag{5}$$

where $\circ$ denotes concatenation and exponentiation denotes repetition.

Proof (sketch). Proof via induction over d (for $d=0$ the statement is always valid). The calculation is similar to the one used in the proof of the Binomial theorem. □

Algorithm 1: Computing $C_{\mathrm{H}}(f)$ given f as a BDD b.

input : $f:\mathbb{B}^n\to\mathbb{B}$ as a BDD b
output: $C_{\mathrm{H}}(f)$

```
1  function compute_C_H(b,l)
2      l_curr ← level(b)
3      d ← l_curr − l
4      if b = 0/1 then
5          return expand(⟨0⟩/⟨1⟩, d)
6      else
7          a ← compute_C_H(low(b), l_curr+1)
8          b ← compute_C_H(high(b), l_curr+1)
9          return expand(a,d) ∘ ⟨0⟩ + ⟨0⟩ ∘ expand(b,d)
10 function expand(v,d)
11     return Σ_{i=0}^{d} (d choose i) · (⟨0⟩^{d−i} ∘ v ∘ ⟨0⟩^{i})
12 return compute_C_H(b,1)
```

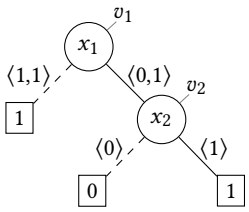


Figure 3: BDD for the function $f(x_1,x_2)=\neg x_1\vee x_1\wedge x_2$. The edges are annotated with the partial C_{H} vectors.

Using Theorem 4.1 allows to easily define the *compute_C_H* method that computes $C_{\mathrm{H}}(f)$ by directly operating on BDDs (see Algorithm 1).

Example 4.2. Consider the function $f(x_1,x_2)=\neg x_1\vee x_1\wedge x_2$ (the corresponding BDD is shown in Figure 3). We compute $C_{\mathrm{H}}(f)=$ *compute_C_H*$(v_1,1)$. The values for a and b (see lines 7 and 8) are $a:=$ *compute_C_H*$(\boxed{1},2)=\langle 1,1\rangle$ and $b:=$ *compute_C_H*$(v_2,2)=\langle 0,1\rangle$. The value b is computed by recursively descending BDD without skipping any level (i.e., $d=0$ in all successive steps). The value a is computed using Theorem 4.1. The value for d is 1 in the call *compute_C_H*$(\boxed{1},2)$ meaning that a level in the BDD has been skipped and, hence, the expansion in line 5 becomes

$$\begin{aligned} expand(\langle 1\rangle,1)&=\binom{1}{0}\left(\langle 0\rangle^1\circ\langle 1\rangle\circ\langle 0\rangle^0\right)+\binom{1}{1}\left(\langle 0\rangle^0\circ\langle 1\rangle\circ\langle 0\rangle^1\right)\\ &=1\cdot\langle 0,1\rangle+1\cdot\langle 1,0\rangle=\langle 1,1\rangle. \end{aligned}$$

This allows to finally compute $C_{\mathrm{H}}(f)$ (see line 9) as

$$C_{\mathrm{H}}(f)=compute\_C_H(b,1)=\langle 1,1\rangle\circ\langle 0\rangle+\langle 0\rangle\circ\langle 0,1\rangle=\langle 1,1,1\rangle.$$

4.3 Symmetrization

The process of the symmetrization of a function f is based on the vectors $T=C_{\mathrm{H}}(f)$ and $D=C_{\mathrm{H}}(f^*)$ and, therefore, does not depend on the representation of either the input function nor the approximated output function.

We briefly recapitulate the idea of the symmetrization process: for every input Hamming weight w decide wether the symmetrized function $\tilde{f}$ should produce a 0 or 1. The decision is made by choosing

Algorithm 2: Determining the value vector.

input : $T = C_{\mathrm{H}}(f), D = C_{\mathrm{H}}(f^*)$
output : Value vector $V_{\tilde{f}}$ of $\tilde{f}$ with minimal Hamming distance $e = \Delta(f,\tilde{f})$ and the error value e itself

1 $V_{\tilde{f}} \leftarrow \langle 0,\ldots,0\rangle$; $e \leftarrow 0$
2 **for** $w \leftarrow 0$ *to* n **do**
3 $zeros \leftarrow \binom{n}{w} - (T[w] + D[w])$
4 **if** $T[w] > zeros$ **then**
5 $V_{\tilde{f}}[w] \leftarrow 1$
6 $e \leftarrow e + zeros$
7 **else**
8 $V_{\tilde{f}}[w] \leftarrow 0$
9 $e \leftarrow e + T[w]$
10 **return** $V_{\tilde{f}}$ *and* e

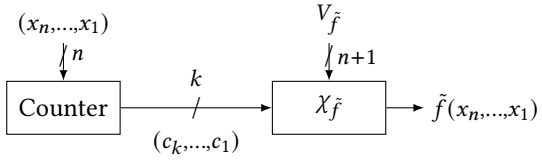


Figure 4: Circuit structure realizing a symmetric function in linear space.

the value that invalidates the least outputs (see Algorithm 2). Note that the number of inputs with Hamming weight w for which f evaluates to 0 is computed by the term $\binom{n}{w} - (T[w] + D[w])$ in line 3.

4.4 Realizing $\tilde{f}$

Given a value vector $V_{\tilde{f}}$, an algorithm for creating a BDD in size n^2 for a symmetric function is presented in [7, Algorithm 2]. The authors further sketch an idea of how to create a circuit in linear size. The idea is to first count the inputs with value 1 and then use the count to select to correct value from the value vector via the characteristic function $\chi_{\tilde{f}}$. The corresponding circuit is visualized in Figure 4.

For the counter, we use the design presented in [11]. The characteristic function is realized using a multiplexer. The $k = \lceil \log_2(n+1) \rceil$ bits are used to select the correct entry of the $n+1$ values in the value vector $V_{\tilde{f}}$.

In our design, the multiplexer is recursively designed using $2:1$ multiplexers. Hence, the circuit realizing $\chi_{\tilde{f}}$ selects from 2^k values. The value 2^k usually is larger than $(n+1)$. This means that we can freely chose the values for the additional $2^k - (n+1)$ outputs in order to allow for an optimal (i.e., smallest) realization of $\chi_{\tilde{f}}$. For this, we developed a fast heuristic that we omit due to page limitations.

Please note that internally a circuit is represented and stored as an AIG (see Section 2 and Figure 1).

4.5 Unbounded Synthesis

Following [7], to approximate multi-output functions $f : \mathbb{B}^n \to \mathbb{B}^m$, it is sufficient to symmetrize the m components f_i using Algorithm 2 and then construct the symmetric function $\hat{f}$ from the symmetric functions $\tilde{f}_i$, i.e., $\hat{f} = (\tilde{f}_1,\ldots,\tilde{f}_n)$.

We observe the following equality

$$\mathrm{wae}_\alpha(f,\hat{f}) = \frac{1}{2^n}\sum_{i=1}^{m} \alpha_i \cdot e_i = \frac{1}{2^n}\sum_{i=1}^{m} \alpha_i \cdot \Delta(f_i,\hat{f}_i),$$

with e_i being computed by Algorithm 2. Obtaining wae_α this way is very important as in general computing arbitrary error metrics between f and $\hat{f}$ is computationally expensive [12], even when using specialized algorithms on efficient function representations such as BDDs [6].

In [7], the authors only used e_i to compute the error rate. This error metric is not well-suited for evaluating arithmetic functions.

Nevertheless, approximating f this way is unsatisfactory as it does not give any control over the resulting error of $\hat{f}$. Hence, we will discuss an approach for approximating the function f using symmetrization while ensuring that $\mathrm{wae}_\alpha(f,\hat{f})$ remains smaller than a given threshold T in the following section.

4.6 Error-bounded Synthesis

To be able to bound the error introduced in the symmetrization process, we sacrifice the total symmetry of $\hat{f}$. Instead, for each $1 \le i \le m$, we decide whether to use f_i or $\tilde{f}_i$ in $\hat{f}$. The choice must ensure that the error threshold T is not violated. In this section we present a generalization of the procedure proposed in [7].

We support not only the error rate as the error metric that is bounded but wae_α for arbitrary $\alpha \in \mathbb{R}_+^m$. Internally, wae_α makes use of $\Delta(f_i,\tilde{f}_i)$, a value that is conveniently already computed by Algorithm 2.

The process is then formalized as follows. Given a Boolean function $f : \mathbb{B}^n \to \mathbb{B}_*^m, f = (f_1,\ldots,f_m)$, its closest symmetrized components $\tilde{f}_i : \mathbb{B}^n \to \mathbb{B}$, an $\alpha \in \mathbb{R}_+^m$ and an error threshold T on $\mathrm{wae}_\alpha(f,\hat{f})$ determine a selection $\sigma \in \mathbb{B}^m$ defining $\hat{f}$ as

$$\hat{f} := \hat{f}_\sigma = (f_{\sigma,1},\ldots,f_{\sigma,m}) \text{ with } f_{\sigma,i} := \sigma_i \tilde{f}_i \vee \overline{\sigma_i} f_i \text{ s.t. } \mathrm{wae}_\alpha(f,\hat{f}_\sigma) \le T \tag{6}$$

In (6), σ_i indicates whether the symmetrized component is used ($\sigma_i = 1$) or not used ($\sigma_i = 0$) in the approximation $\hat{f}$.

There might be many σ's satisfying (6). Therefore, we need a means to constrain σ so that we select a "good" σ. The question is how to measure the quality of σ. For this, we define a *cost function* to be a function $C : \mathbb{B}^m \to \mathbb{R}_+$ that measures the cost (e.g., size of the resulting approximation $\hat{f}_\sigma$) of the selection σ. This allows us to formulate the following minimization problem

$$\sigma^* = \mathrm{argmin}_{\sigma \in \mathbb{B}^m} C(\sigma) \quad \text{subject to } \mathrm{wae}_\alpha(f,\hat{f}_\sigma) \le T \tag{7}$$

Without further knowledge about C, determining an optimal solution of (7) might require to test all 2^m possible values for σ. We relax the exact optimization problem (7) and approximate it by trying to find a σ that minimizes the sum of the costs of the individual components:

$$\sigma^* = \mathrm{argmin}_{\sigma \in \mathbb{B}^m} \sum_{i=1}^{m} C_i(\sigma_i) \quad \text{subject to } \mathrm{wae}_\alpha(f,\hat{f}_\sigma) \le T \tag{8}$$

Using individual cost function allows to formulate the following theorem.

Theorem 4.3. *Consider cost functions* $C:\mathbb{B}^m \to \mathbb{R}_+$ *and* $C_i:\mathbb{B}\to\mathbb{R}_+$ *for* $1\le i\le m$ *and a constant* $\beta\in\mathbb{R}_+$ *with the property*

$$\beta\cdot C(\sigma)\ge\sum_{i=1}^{m}C_i(\sigma_i)\ge C(\sigma). \tag{9}$$

If σ^* *is a solution to the exact optimization problem* (7) *and* $\sigma^\dagger$ *is solution to the approximated optimization problem* (8)*, then the following inequality holds*

$$C(\sigma^*)\ge\frac{1}{\beta}\cdot C(\sigma^\dagger)$$

The approximated solution of (8) is, therefore, worse than the optimal solution by at most a factor of β.

Proof. Simple calculation; omitted due to page limitations. □

Example 4.4. When aiming to symmetrize f as much as possible, the choice for C is to count the non-symmetrized components, i.e., $C(\sigma)=\sum_{i=1}^{m}\overline{\sigma_i}$. Counting the selection individually, i.e., $C_i(\sigma_i)=\overline{\sigma_i}$, for solving the approximated problem (8), Theorem 4.3 applies with $\beta=1$. For this particular choice of C_i, a solution to (8) also solves the original problem (7). In case of $\text{wae}_\alpha=\text{er}$, this reduces to the procedure presented in [7].

Example 4.5. One can also aim to directly minimize the size of the representation of $\hat{f}$ by choosing $C(\sigma)=\#BDD(\hat{f}_\sigma)$ or $C(\sigma)=\#AIG(\hat{f}_\sigma)$. When choosing $C_i(\sigma_i)=\#BDD(\hat{f}_{\sigma,i})/\#AIG(\hat{f}_{\sigma,i})$, Theorem 4.3 applies with $\beta=m$.

This can easily be seen as the size of each component $f_{\sigma,i}$ can be as most the size of f_σ. This yields the inequality

$$\sum_{i=1}^{m}\#BDD(f_{\sigma,i})\le\sum_{i=1}^{m}\#BDD(f_\sigma)=m\cdot\#BDD(f_\sigma).$$

The other inequality in (9) follows immediately from the fact that f_σ can not be bigger than the sum of all its components' sizes. The same argument holds for the AIG representation of f_σ.

In order to actually solve the approximated optimization problem (8), we re-formulate it as a 0/1 Knapsack problem [13]. The knapsack problem is then given as

$$\text{maximize}\sum_{i=1}^{m}\sigma_i p_i\quad\text{subject to}\sum_{i=1}^{m}\sigma_i e_i\le T \tag{10}$$

with $p_i=C_i(0)-C_i(1)$ and $e_i=\frac{\alpha_i}{2^n}\Delta(f_i,\tilde{f}_i)$. The "values" p_i encode the cost gains by choosing the approximated function while the "weights" e_i encode the introduced error wae_α.

5 EXPERIMENTAL RESULTS

In this section, we apply the methodology introduced in Section 4 on a broad set of benchmarks, namely

(1) functions form the EPFL benchmark suite [14] and benchmarks from the ESPRESSO tool [15] selected to match the choice in [7] (networks),
(2) adders and multipliers of different bit width (add, mult),
(3) multiplication of k pairs of n bit numbers with a subsequent addition step (mac) and
(4) maximally asymmetric functions of different bit widths (asymm)

Benchmarks (2) – (4) are generated using ABC [16]. The algorithm to generate asymm has been taken from [17]. All benchmarks are converted to both, BDDs and AIGs. These BDDs/AIGs are then optimized for size to establish a base line to compare the approximated functions against.

Benchmarks (1) and (4) are evaluated using the error rate as the outputs do not represent numbers while benchmarks (2) and (3) are evaluated using the nwae_α metric as an upper bound on ace. We chose nwae over wae in order to also be able to compare results of different bit width within one set of functions.

All benchmarks are optimized for the number of symmetric components as well as the BDD/AIG size for both, the bounded and unbounded case. We chose 5% as the error threshold as this allows a direct comparison with [7] on benchmark set (1). The knapsack problem has been solved approximately following [13], i.e., iteratively choosing the component with the highest value of p_i/e_i as long as (a) $p_i>0$ and (b) the error is below the threshold T.

The proposed framework has been implemented in C++ within ABC [16]. All experiments have been run on a Linux (Ubuntu 23.04) machine with an Intel Core i7-8550U CPU and 24G of main memory.

In the first set of experiments, we approximately synthesize the networks benchmark set and compare the results with those of [18, Table 4] (see Table 1). When optimizing the number of symmetric components selected (column "# sym. comp."), our framework produces slightly worse results than [7] (unfortunately, the related work does not report the achieved average error rate when bounding it at 5%). This is due to the fact that we start with size-optimized, i.e., smaller, benchmarks. When directly optimizing for BDD or AIG size (columns "# BDD" and "# AIG") instead of simply counting the number of symmetrized components, the proposed framework clearly outperforms the approach of [7]. It should be noted that, on average, only half of the allowed error rate of 5% is reached in the synthesized design (see the columns "Øer" in the BDD/AIG parts of Table 1). This clearly shows that using BDD/AIG size results in better overall results. All approximations have been computed in negligible runtime; only the `i2c` benchmark took ∼1.6s in total.

In the next set of benchmarks, we investigate the arithmetic functions add, mult and mac. We synthesize n bit adders with n up to 256 with a stride of 16, n bit multipliers from $n=1$ to $n=13$ and MAC functions where k ranges from 2 to 5 and n from 1 up to 5, 5, 4, and 3 for $k=2$, $k=3$, $k=4$ and $k=5$, respectively. The synthesis results for the bounded approximate synthesis with a threshold of $T=5$ for nwae_α are shown in Table 2. We do not report unbounded results or results for the total number of symmetric components as the optimization goal. These results are of low quality (see the short discussion in Section 4.5)

Negative gains come from shared nodes. Nodes shared between components are not (and can not be) taken into account when deciding whether to replace the component by its symmetrized version. The symmetric component $f_{\sigma,i}$ might then account for almost $\#f_{\sigma,i}$ new nodes when no sharing is possible. This is in line with the theoretical findings of Theorem 4.3 (see also Example 4.5).

In case of the adder circuits, all symmetrized components are larger than the original ones. Hence, no symmetrization and, consequently, no approximation is carried out. For the AIGs this is expected as per construction of the circuit. We did not yet find an explanation for why this also holds true in the BDD case.

The execution time of all benchmarks is dominated by the creation of the symmetric components and, except for the adders, is entirely negligible. In the adder case, the creation of the symmetric BDD components starts to exceed one minute with an input bit width of 112, reaching a ≈1 hour for 256 bits. This is in line with the corresponding algorithm being in $O(n^5)$ [7].

Table 1: Comparison of average gains and average error rate for different optimization criteria for the unbounded and bounded approximate synthesis.

	[7]		proposed symmetry-based approximate synthesis framework (SAS)					
	# sym. comp.		# sym. comp.		# BDD		# AIG	
	∅ gain	∅ er	∅ gain (BDD/AIG)	∅ er	∅ gain	∅ er	∅ gain	∅ er
unbounded	58.6%	14%	29%/58.4%	13.9%	66.7%	8.6%	66.3%	8.1%
bounded at 5%	36.3%	n/a	21.2%/25.2%	2.8%	47.1%	2.6%	45.0%	2.4%

Table 2: Synthesis results for the bounded synthesis with nwae ≤ 5 for arithmetic functions.

	# BDD		# AIG	
benchmark	∅ gain	∅ er	∅ gain	∅ er
ADD	0%	0%	0%	0%
MULT	48.6%	2.9%	−22.1%	3.2%
MAC	25.8%	3.3%	−35.1%	2.1%

Table 3: Synthesis result for the unbounded synthesis for selected maximally asymmetric functions.

#PI	# BDD	# AIG	er	time (s)
4	85.71%	100.00%	43.75%	0.02
8	85.71%	96.89%	49.61%	0.04
12	92.36%	97.40%	49.95%	0.28
16	99.79%	99.79%	50.00%	6.16
18	99.72%	99.90%	50.00%	22.28

In the last set of benchmarks we investigate the synthesis behaviour for maximally asymmetric functions (see [17]). We expect that these functions are the most difficult to approximate as they are "as far away" from symmetric functions as possible. We generated functions for 1 to 18 inputs. Selected and representative synthesis results are shown in Table 3. Possible gains in BDD/AIG size are tremendous but with larger input bit width, as expected, the error rate approaches 50%. This shows that the proposed framework is best applied when the function to be approximated is not too asymmetric when the introduced error is important. Only the unbounded synthesis results are shown as for the bounded synthesis, the threshold would have to be set to $\approx 50\%$ for the symmetrized function to be selected. Almost 100% of the execution time is spend in generating the symmetric component AIGs, i.e., when optimizing for BDD size, the execution time can be neglected.

All scripts necessary to reproduce the results are available on GitHub.[2]

6 CONCLUSIONS AND OUTLOOK

We presented SAS, a framework for Symmetry-based Approximate Synthesis and evaluated its applicability and efficiency on a wide range of benchmarks, including arithmetic functions as well as maximally asymmetric functions as the class of the most difficult to synthesize functions. As expected, SAS performs best when the approximated functions are not too asymmetric.

The proposed generalized error metric wae_α allows SAS to be applied to many different classes of functions, including arithmetic ones by serving as an upper bound for the average case error ace. In future work, we will evaluate how tight this bound actually is.

[2] https://github.com/keszocze/sas/

REFERENCES

[1] N. Zhu, W. L. Goh, W. Zhang, K. S. Yeo, and Z. H. Kong. "Design of Low-Power High-Speed Truncation-Error-Tolerant Adder and Its Application in Digital Signal Processing". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 18.8 (2010).

[2] J. Han and M. Orshansky. "Approximate Computing: An Emerging Paradigm for Energy-Efficient Design". In: *European Test Symposium*. 2013.

[3] J. Echavarria, S. Wildermann, and J. Teich. "Design Space Exploration of Multi-output Logic Function Approximations". In: *International Conference On Computer Aided Design*. 2018.

[4] M. Češka, J. Matyáš, V. Mrazek, L. Sekanina, Z. Vasicek, and T. Vojnar. "Approximating Complex Arithmetic Circuits with Formal Error Guarantees: 32-Bit Multipliers Accomplished". In: *International Conference On Computer Aided Design*. 2017.

[5] S. Shirinzadeh, M. Soeken, D. Große, and R. Drechsler. "An Adaptive Prioritized E-preferred Evolutionary Algorithm for Approximate BDD Optimization". In: *Conference on Genetic and Evolutionary Computation*. 2017.

[6] O. Keszocze. "BDD-based Error Metric Analysis, Computation and Optimization". In: *IEEE Access* 10 (2022).

[7] A. Bernasconi, V. Ciriani, and T. Villa. "Exploiting Symmetrization and D-Reducibility for Approximate Logic Synthesis". In: *IEEE Transactions on Computers* 71.1 (2022).

[8] M. Heap. "On the Exact Ordered Binary Decision Diagram Size of Totally Symmetric Functions". In: *Journal of Electronic Testing* 4.2 (1, 1993).

[9] V. Mrazek, R. Hrbacek, Z. Vasicek, and L. Sekanina. "EvoApprox8B: Library of Approximate Adders and Multipliers for Circuit Design and Benchmarking of Approximation Methods". In: *Design, Automation and Test in Europe*. 2017.

[10] G. Fey and R. Drechsler. "Utilizing BDDs for Disjoint SOP Minimization". In: *Midwest Symposium on Circuits and Systems*. Vol. 2. 2002.

[11] E. Swartzlander. "Parallel Counters". In: *IEEE Transactions on Computers* C-22.11 (1973).

[12] O. Keszocze, M. Soeken, and R. Drechsler. "The Complexity of Error Metrics". In: *Information Processing Letters* 139 (2018).

[13] S. Sahni. "Approximate Algorithms for the 0/1 Knapsack Problem". In: *Journal of the Association for Computing Machinery* 22.1 (1975).

[14] L. Amarù, P.-E. Gaillardon, and G. De Micheli. "The EPFL Combinational Benchmark Suite". In: *International Workshop on Logic & Synthesis*. 2015.

[15] S. Yang. *Logic Synthesis and Optimization Benchmarks User Guide Version 3.0*. 15, 1991.

[16] R. Brayton and A. Mishchenko. "ABC: An Academic Industrial-Strength Verification Tool". In: *Computer Aided Verification*. Ed. by T. Touili, B. Cook, and P. Jackson. 2010.

[17] S. Nagayama, T. Sasao, and J. T. Butler. "On Decision Diagrams for Maximally Asymmetric Functions". In: *International Symposium on Multiple-Valued Logic*. 2022.

[18] A. Bernasconi, V. Ciriani, and T. Villa. "Approximate Logic Synthesis by Symmetrization". In: *Design, Automation and Test in Europe*. 2019.