

Provable Guarantees in Approximate Synthesis

Kushagra Gupta
IIT Delhi

Priyanka Golia
IIT Delhi

Subhajit Roy
IIT Kanpur

Kuldeep S Meel
University of Toronto
Georgia Institute of Technology

Abstract—Automated synthesis techniques generate systems—such as functions or circuits—that provably satisfy a formal specification. Traditional synthesis frameworks often adopt an all-or-nothing approach: either the system satisfies all constraints, or synthesis fails. However, in many practical settings, such strict completeness is either infeasible or too costly to achieve, especially in terms of resources like time, memory, or circuit area. This work addresses such scenarios by moving beyond the all-or-nothing paradigm. We propose a novel synthesis framework that distinguishes between hard constraints, which must be strictly satisfied, and soft constraints, which may be relaxed. The goal is to synthesize systems that provably satisfy all hard constraints while achieving a user-defined threshold of satisfiability on the soft constraints. We quantify this relaxation using a satisficing measure, such as accuracy—i.e., the proportion of inputs for which the system satisfies all constraints.

Our approach integrates AI-based methods to generate candidate systems and automated reasoning techniques to ensure formal guarantees. Through extensive experiments, we show that our framework significantly reduces synthesis time compared to traditional approaches. Moreover, the synthesized systems (e.g., circuits) tend to be smaller, connecting our method naturally to the domain of approximate circuit synthesis. Unlike existing approximate synthesis techniques, our framework provides formal guarantees on both correctness (for hard constraints) and quality (for soft constraints).

I. INTRODUCTION

Automated synthesis techniques construct systems—such as functions or circuits—that provably satisfy formal specifications. Despite their computational hardness, the past decade has seen remarkable progress in synthesis methods [1]–[7]. Still, most frameworks follow an all-or-nothing paradigm: either every constraint is satisfied, or synthesis fails. Such completeness is often infeasible or prohibitively costly in runtime or circuit area.

To overcome this, prior work explored approximate synthesis, trading accuracy for reduced resource usage [8]. Much prior work focuses on approximate logic synthesis, which rewrites RTL/netlists into smaller gate-level designs under error constraints [9]–[15]. Yet two key limitations remain: (i) guarantee-preserving methods fail to scale [16], and (ii) scalable methods lack formal guarantees on user-specified logical constraints beyond error bounds—which is problematic in safety-critical applications [17]. Note that accuracy thresholds over a test suite alone cannot ensure correctness for critical system components.

We propose to close this gap by introducing a hybrid approach that **enforces provable guarantees on critical**

components while providing quantifiable quality elsewhere. This balance reduces runtime, memory, or area, while ensuring correctness where it matters most.

Two examples illustrate the idea. (i) In on-chip router placement, adding more routers improves communication throughput in a Network-on-Chip (NoC), but also inflates chip area and power consumption [18], [19]. A designer may require exact guarantees for one routing direction while tolerating approximation in others. (ii) In a comparator deciding whether $a + b \geq 8$ for two 4-bit integers, ensuring correctness of the most significant bits suffices, even if lower bits are approximated.

These scenarios highlight the value of “approximate-but-correct” synthesis when exact solutions are infeasible. Designers can enforce safety-critical properties as hard constraints while relaxing less critical ones—such as arithmetic precision—to save area, power, or latency.

We formalize this perspective by distinguishing: hard constraints, which must always hold; and soft constraints, which may be relaxed subject to a satisficing measure (e.g., accuracy, area, or latency).

Our problem statement is domain-agnostic:

SatisficingSynth: Given hard constraints, soft constraints, a satisficing measure, and a threshold, synthesize a system that

- provably satisfies all hard constraints, and
- achieves a satisficing measure on soft constraints above the given threshold.

This formulation is flexible and application dependent: the user specifies hard and soft constraints, selects a satisficing measure, and set threshold reflecting design goals. In Section III, we instantiate it for Boolean circuit synthesis.

We instantiate this framework in SAMYAK, targeting approximately correct Boolean functions. For instance, accuracy may be defined as the proportion of inputs satisfying both hard and soft constraints. To our knowledge, SAMYAK is the first approximate synthesis framework that synthesizes Boolean functions (or circuits) with *provable guarantees on selected constraints or bits* while quantifying performance across the input space.

At a high level, SAMYAK works in three phases: (i) learn a candidate function from hard and soft constraints (e.g., via decision trees); (ii) validate against hard constraints and repair conflicts using unsatisfiable cores; (iii) verify the satisficing threshold, iteratively repairing the candidate to improve soft-constraint performance without violating hard constraints.

We evaluate SAMYAK on logic/control benchmarks from the EPFL Combinational Benchmark Suite [20] (specifically the XY-lookahead router [21], [22]), bit-level benchmarks [23], and non-deterministic arithmetic tasks such as factorization and decomposability [24], [25]. SAMYAK consistently outperforms exact synthesis in runtime while meeting satisficing thresholds above 80% in nearly all cases. Beyond accuracy, it also yields smaller circuits: on the router-array benchmarks, SAMYAK reduces area by at least 10%, with over **20%** reduction in 77% of configurations. It also scales well, showing that relaxing soft constraints while enforcing critical guarantees can simultaneously improve secondary metrics such as area.

II. NOTATIONS AND PRELIMINARIES

We use lowercase letters (with or without subscripts) to denote propositional variables and upper case letters to denote a subset of variables. A *literal* is a Boolean variable or its negation. A *clause* is disjunction of literals. A formula F is in Conjunctive Normal Form (CNF) if it is represented as conjunction of clauses. We use $Vars(F)$ to denote the set of variables of formula F .

A *satisfying assignment* of F maps $Vars(F)$ to $\{0,1\}$ such that F evaluates to true. We write $\sigma \models F$ if σ is a satisfying assignment of F . A formula is satisfiable if such an assignment exists; otherwise, it is unsatisfiable. Given $P \subseteq Vars(F)$, let $\sigma_{\downarrow P}$ denote the assignment of variables restricted to P . We denote the set of all satisfying assignments of F by $sol(F)$, and use $sol(F)_{\downarrow P}$ to indicate the projection of $sol(F)$ on P .

The problem of propositional model counting is to compute the number of satisfying assignments of F , that is, compute $|sol(F)|$, denoted by $ModelCount(F)$. Given a formula F and a projection set P , the projected model counting problem is to compute $|sol(F)_{\downarrow P}|$. We use $ModelCount(F_{\downarrow P})$ to represent projected model count of F on P .

Example 1: Let us consider $X = \{x_1, x_2\}$, $P = \{x_1\}$, and $F = x_1 \vee x_2$. One satisfying assignment of F is $\sigma = \langle x_1 \leftrightarrow 0, x_2 \leftrightarrow 1 \rangle$. So, $\sigma_{\downarrow P} = \langle x_1 \leftrightarrow 0 \rangle$. Then, $sol(F) = \{\langle x_1 \leftrightarrow 0, x_2 \leftrightarrow 1 \rangle, \langle x_1 \leftrightarrow 1, x_2 \leftrightarrow 0 \rangle, \langle x_1 \leftrightarrow 1, x_2 \leftrightarrow 1 \rangle\}$, $ModelCount(F) = 3$, $sol(F)_{\downarrow P} := \{\langle x_1 \leftrightarrow 0 \rangle, \langle x_1 \leftrightarrow 1 \rangle\}$, and $ModelCount(F_{\downarrow P}) = 2$.

The uniform sampling problem takes a formula F and integer K and returns K satisfying assignments sampled uniformly at random from the solution space of F . For an unsatisfiable formula F , an *UnsatCore* is a subset of clauses of F for which there does not exist a satisfying assignment. There could exist multiple such *UnsatCore* of a formula. A partial MaxSAT solver takes hard and soft constraints as input and returns a satisfying assignment that satisfies all hard constraints and maximally satisfies soft constraints. We use $\varphi_H(X, Y)$ and $\varphi_S(X, Y)$ to denote set of hard and soft constraints over set of inputs X and outputs Y . Let $F(X) := \langle f_1(X), \dots, f_m(X) \rangle$ be a vector of Boolean functions over X , where $m = |Y|$. We use the following to denote logical equivalences: (i) $(Y \leftrightarrow F(X)) := (y_1 \leftrightarrow f_1(X), \dots, y_m \leftrightarrow f_m(X))$, (ii) $(X \leftrightarrow \sigma_{\downarrow X}) := (x_1 \leftrightarrow \sigma_{x_1}, \dots, x_n \leftrightarrow \sigma_{x_n})$.

III. OVERVIEW

In this section, we provide a high-level overview of a framework to mitigate the problem of SatisficingSynth. SatisficingSynth deals with a satisficing measure, which is defined as $SM()$ that takes two inputs, (i) a specification (say, $\psi(X, Y)$), and (ii) a function (say, $G(X)$) to compute the following:

$$SM(\psi(X, Y), G(X)) := \frac{ModelCount((\psi(X, Y) \wedge (Y \leftrightarrow G(X)))_{\downarrow X})}{ModelCount(\psi(X, Y)_{\downarrow X}} \quad (1)$$

Informally, $SM()$ denotes the accuracy—the ratio of input valuations (i.e., assignments to X) for which the synthesized system G satisfies its specification ψ , to the total number of possible input valuations. Note that unlike accuracy defined over sampled test cases, this definition is exhaustive, ensuring coverage of all cases.

Now, let us define the problem under consideration, SatisficingSynth, formally as follows:

Given (i) hard constraints $\varphi_H(X, Y)$, (ii) soft constraints $\varphi_S(X, Y)$, and (iii) a satisficing threshold ϵ , where X is a set of inputs and Y is a set of outputs, the objective is to synthesize a system $F(X)$ such that following holds:

- $\forall X (\exists Y \varphi_H(X, Y) \leftrightarrow \varphi_H(X, F(X)))$,
- $SM(\varphi_S(X, Y), F(X)) \geq \epsilon$.

Informally, the synthesized functions F must satisfy all hard constraints $\varphi_H(X, Y)$, while the satisficing measure $SM()$ —computed from the soft constraints $\varphi_S(X, Y)$ and $F(X)$ as in Equation 1—must exceed a predefined threshold. In other words, F provides guaranteed satisfaction of hard constraints and achieves at least threshold-level accuracy on soft constraints.

Towards synthesizing such a function F , we propose a framework called SAMYAK. The overview of SAMYAK is shown in Figure 1.

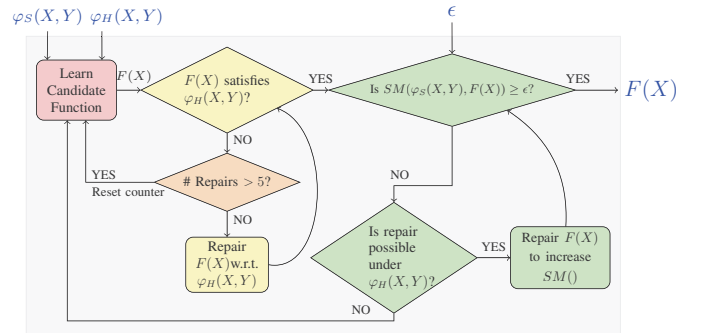


Fig. 1. SAMYAK for synthesizing $F(X)$ that satisfies $\varphi_H(X, Y)$ and satisficing measure $SM(\varphi_S(X, Y), F(X))$ is greater than threshold ϵ .

As shown in Figure 1, SAMYAK first learns a candidate function $F(X)$ from $\varphi_S(X, Y)$ and $\varphi_H(X, Y)$. It then checks whether the hard constraints are satisfied; if not, the candidate is repaired. If repeated repairs fail (set threshold as 5), SAMYAK relearns a new candidate and checks again. Once the hard constraints are satisfied, SAMYAK evaluates whether

the satisficing measure $SM()$ exceeds the threshold ε . If so, $F(X)$ is returned. Otherwise, SAMYAK attempts either to repair the candidate (without violating hard constraints) to improve $SM()$, or to learn a new candidate system.

We now provide a high-level description of different components of SAMYAK to highlight the technical challenges.

a) Learn Candidate $F(X)$: Different underlying synthesizers can be used to generate the candidate function $F(X)$. We leverage advances in AI/ML to learn $F(X)$ by first generating data through sampling satisfying assignments of relation $\varphi(X, Y) = \varphi_S(X, Y) \cup \varphi_H(X, Y)$. A decision tree-based approach is then used to learn a candidate function over the produced data [26], [27]. This approach treats valuations of X as features and valuations of each $y \in Y$ as labels to learn a decision tree for each output variable, and $F(X)$ is constructed as a disjunction of the paths in the tree where the leaf node is **1**. Alternatively, approximate synthesis engines can generate an initial $F(X)$ [8].

b) Does $F(X)$ satisfy $\varphi_H(X, Y)$?: Once a candidate $F(X)$ is obtained, we verify hard-constraint satisfaction using a SAT solver by checking the satisfiability of $V(X, Y, Y') := \varphi_H(X, Y) \wedge \neg\varphi_H(X, Y') \wedge (Y' \leftrightarrow F(X))$.

A key question regarding $V(X, Y, Y')$ is: why is the first term needed? Specifically, one might ask why we can't simply check whether there exists an input for which the synthesized system violates the hard constraints, i.e., $\neg\varphi_H(X, Y') \wedge (Y' \leftrightarrow F(X))$. Consider the case where the specification is integer factorization, and the goal is to compute non-trivial factors of a given input. If the input is a prime number, the specification itself is not satisfiable—no non-trivial factors exist. In such cases, checking whether the synthesized function satisfies the specification on prime inputs would be meaningless, since the specification is false for those inputs. To avoid such cases, we restrict the check to inputs for which the hard constraint is satisfiable. That is, we verify whether, for all input valuations where $\varphi_H(X, Y)$ holds, the synthesized function $F(X)$ also satisfies the hard constraint. Therefore, if $V(X, Y, Y')$ is `unsat`, then $F(X)$ satisfies the hard constraints; otherwise, the satisfying assignment provides a counterexample for repair.

c) Repair $F(X)$ with respect to $\varphi_H(X, Y)$: Let $\sigma \models V(X, Y, Y')$ be a counterexample for which SAMYAK must repair the candidate function $F(X)$. We aim to identify the *reason* why $F(X)$ fails to satisfy the hard constraints under the input valuation $\sigma_{\downarrow X}$. To this end, SAMYAK employs a proof-guided strategy to repair $F(X)$, leveraging the unsatisfiability core of the formula $M(X, Y) := \varphi_H(X, Y) \wedge (Y \leftrightarrow F(X)) \wedge (X \leftrightarrow \sigma_{\downarrow X})$ to capture the underlying cause of the violation.

The unsatisfiability core (UNSAT core) is the subset of hard constraints violated by the synthesized function. To address these violations, SAMYAK either strengthens or weakens the candidate function $F(X)$. It extracts unit clauses from the `UnsatCore` of $M(X, Y)$ —specifically those over input literals X —to construct a subformula that guides repairs and updates $F(X)$. If a candidate undergoes too many repairs, SAMYAK re-learns it with an enlarged sample set. A counter r_y tracks repair attempts: if r_y is below a predefined threshold,

localized proof-guided repairs are applied; once r_y exceeds the threshold, SAMYAK re-learns $F(X)$, resets $r_y := 0$, and increases the sample set. In our implementation, the threshold is set to 5.

d) Is $SM(\varphi_S(X, Y), F(X)) \geq \varepsilon$?: Once $F(X)$ is established to satisfy the hard constraints, SAMYAK computes the satisficing measure $SM(\varphi_S(X, Y), F(X))$ using Equation 1. It then checks whether this measure is lower-bounded by the given threshold ε .

e) Is repair possible?: If $SM(\varphi_S(X, Y), F(X)) < \varepsilon$, we seek a counterexample—an input where $F(X)$ satisfies hard constraints but violates soft constraints, and where some output Y satisfies the soft constraints. SAMYAK checks satisfiability of $U(X, Y, Y') : \varphi_S(X, Y) \wedge \neg\varphi_S(X, Y') \wedge \varphi_H(X, Y') \wedge (Y' \leftrightarrow F(X))$.

If it turns out to be `UNSAT`, no repair exists without violating hard constraints. In contrast if it is `SAT`, we obtain counterexample π for repair to improve $SM()$.

f) Repair $F(X)$ to increase $SM()$: SAMYAK now obtains a counterexample π such that $\pi \models U(X, Y, Y')$. Similar to the repair process for hard constraint violations, SAMYAK employs an `UNSAT` core-driven strategy to repair $F(X)$ and improve the satisficing measure. To initiate this, SAMYAK makes a SAT query on formula: $N(X, Y) := \varphi_S(X, Y) \wedge \varphi_H(X, Y) \wedge (Y \leftrightarrow F(X)) \wedge (X \leftrightarrow \pi_{\downarrow X})$.

There may be multiple `UNSAT` cores for the formula, leading to various possible ways to repair $F(X)$. Since the goal is to improve the satisficing measure with respect to soft constraints, SAMYAK enumerates different `UNSAT` cores of the formula $N(X, Y)$ and performs proof-guided repair. First, it must be ensured that a proposed repair does not violate the hard constraints. To do so, SAMYAK issues a satisfiability check using Formula $V(X, Y, Y')$ ((b) in overview). for each repair candidate. Any repair for which Formula $V(X, Y, Y')$ is `SAT` is discarded. Next, SAMYAK evaluates the satisficing measure for the remaining repair options. Following a greedy strategy, it selects the repair that has highest satisficing measure among all repair options.

IV. ALGORITHM

Based on the above overview, we now briefly present the three key algorithms that formalize SAMYAK's synthesis and repair procedures in Algorithms 1, 2, and 3.

Algorithm 1 assumes access to following subroutines:

- 1) `LearnCandidate`: It takes $\varphi_S(X, Y), \varphi_H(X, Y)$ as inputs and outputs the initial candidate function $F(X)$. SAMYAK uses a decision tree based method to learn the candidate function using an approach proposed in [26], leveraging advances in AI/ML. SAMYAK first uses a constraint sampler to uniformly generate satisfying assignments from $\varphi(X, Y) = \varphi_S(X, Y) \cup \varphi_H(X, Y)$. SAMYAK then considers the valuation of X on those assignments as features and valuations of each y of Y as label to learn a decision tree for each y . Candidate function $F(X)$ is a disjunction of each path in learned decision tree with leaf node **1**. Decision tree-based

Algorithm 1: SAMYAK($\varphi_S(X, Y), \varphi_H(X, Y), \varepsilon$)

```

1  $F(X) \leftarrow \text{LearnCandidate}(\varphi_S(X, Y), \varphi_H(X, Y))$ 
2 repeat
3    $\text{ret}, \sigma \leftarrow$ 
4    $\text{CheckSAT}(\varphi_H(X, Y) \wedge \neg\varphi_H(X, Y') \wedge (Y' \leftrightarrow F(X)))$ 
5   if  $\text{ret} = \text{SAT}$  then
6     if  $r_y \geq 5$  then
7        $F(X) \leftarrow \text{LearnCandidate}(\varphi_S(X, Y), \varphi_H(X, Y))$ 
8        $r_y \leftarrow 0$ 
9     else
10       $F(X) \leftarrow \text{RepairHC}(\varphi_H(X, Y), F(X), \sigma)$ 
11       $r_y \leftarrow r_y + 1$ 
12 until  $\text{ret} = \text{UNSAT}$ 
13 while  $\text{True}$  do
14   if  $SM(\varphi_S(X, Y), F(X)) < \varepsilon$  then
15      $\text{res}, \pi \leftarrow \text{CheckSAT}(\varphi_S(X, Y) \wedge \neg\varphi_S(X, Y') \wedge$ 
16      $\varphi_H(X, Y') \wedge (Y' \leftrightarrow F(X)))$ 
17     if  $\text{res} = \text{SAT}$  then
18        $F(X), \text{pos} \leftarrow$ 
19        $\text{RepairGM}(\varphi_S(X, Y), \varphi_H(X, Y), F(X), \pi)$ 
20     if  $\text{res} = \text{UNSAT}$  or  $\text{pos} = \text{False}$  then
21        $\varphi_S(X, Y) \leftarrow \varphi_S(X, Y) \wedge (Y \neq F(X))$ 
22       Goto line 1
23 else
24   return  $F(X)$ 

```

learning of candidate function is also employed in the state-of-the-art function synthesizers [27]. Different underlying synthesizers can be used to generate $F(X)$, such as approximate synthesis engines can be used [8].

- 2) CheckSAT: It takes a formula as input and returns the outcome of a satisfiability check on the formula. If SAT, it returns the outcome as SAT and a satisfying assignment; if UNSAT, returns outcome as UNSAT and empty list.
- 3) RepairHC: It takes the hard constraints $\varphi_H(X, Y)$, the candidate function $F(X)$ and a counterexample σ as inputs, and returns a candidate function repaired with respect to the hard constraints $\varphi_H(X, Y)$. RepairHC is discussed in detail in Algorithm 2.
- 4) RepairGM: It takes $\varphi_S(X, Y)$, $\varphi_H(X, Y)$, $F(X)$ and a counterexample π as inputs. If there exists a repaired candidate function with respect to $\varphi_S(X, Y)$ that (i) does not violate $\varphi_H(X, Y)$, and (ii) increases $SM()$, RepairGM returns repaired $F(X)$ and True. Else, it returns original $F(X)$ and False. RepairGM is discussed in detail in Algorithm 3.

Algorithm 1 presents the main SAMYAK routine. It learns a candidate $F(X)$ (line 1), repairs hard constraint violations in a loop (lines 11-9), and iteratively improves satisficing measures (lines 14-16).

Algorithm 2 presents hard constraint repair using UNSAT cores (line 1) to strengthen or weaken $F(X)$ (lines 4-6). Algorithm 2 assumes access to FindCoreUnsat, which extracts the unsatisfiable core and returns unit clauses C in the unsatisfiable core. It employs a proof-guided strategy, leveraging the unsatisfiability core C of the formula discussed in (c) of overview.

Algorithm 3 presents soft constraint repair by enumerating UNSAT cores (line 1) and greedily selecting the repair with maximum $SM()$ (line 11). Algorithm 3 assumes access to a subroutine CoreEnum which takes an UNSAT formula and

Algorithm 2: RepairHC($\varphi_H(X, Y), F(X), \sigma$)

```

1  $C \leftarrow \text{FindCoreUnsat}(\varphi_H(X, Y) \wedge (Y \leftrightarrow F(X)) \wedge (X \leftrightarrow \sigma_{\downarrow X}))$ 
2  $\beta \leftarrow \bigwedge_{l \in C} l$ 
3 if  $F(X \leftrightarrow \sigma_{\downarrow X}) == 1$  then
4    $F(X) \leftarrow F(X) \wedge \neg\beta$ 
5 else
6    $F(X) \leftarrow F(X) \vee \beta$ 
7 return  $F(X)$ 

```

Algorithm 3: RepairGM($\varphi_S(X, Y), \varphi_H(X, Y), F(X), \pi$)

```

1  $Clist \leftarrow \text{CoreEnum}(\varphi_S(X, Y) \wedge \varphi_H(X, Y) \wedge (Y \leftrightarrow$ 
2    $F(X)) \wedge (X \leftrightarrow \pi_{\downarrow X}), K)$ 
3   /* Enumeration of K unsat cores */
4  $Sys \leftarrow \langle \emptyset, \emptyset \rangle$ 
5 for each  $C \in Clist$  do
6    $\beta \leftarrow \bigwedge_{l \in C} l$ 
7    $rF(X) \leftarrow \text{ite}((F(\pi_{\downarrow X}) == 1), F(X) \wedge \neg\beta, F(X) \vee \beta)$ 
8    $\text{ret}, \sigma \leftarrow$ 
9    $\text{CheckSAT}(\varphi_H(X, Y) \wedge \neg\varphi_H(X, Y') \wedge (Y' \leftrightarrow rF(X)))$ 
10  if  $\text{ret} = \text{UNSAT}$  then
11     $Sys \leftarrow Sys.add(\langle SM(\varphi_S(X, Y), rF(X)), rF(X) \rangle)$ 
12 if  $Sys = \langle \emptyset, \emptyset \rangle$  then
13   return  $F(X), \text{False}$ 
14  $F(X) \leftarrow \text{FindRepair}(Sys)$ 
15 return  $F(X), \text{True}$ 

```

enumerates up to its K unsatisfiable cores. For each core, let C denote the list of unit clauses that constitute it; CoreEnum adds each C to a global list C_{list} and returns it.

Algorithmic Optimizations

For $|Y| > 1$, we learn each y_i as a function of X and $Y \setminus y_i$, ensuring no cyclic dependencies: if f_i depends on y_j , then f_j cannot depend on y_i . Let $y_i \prec_d y_j$ denote such a dependency. A system vector F is valid if \prec_d forms a partial order over $y_1, \dots, y_{|Y|}$; its linear extension *TotalOrder* guides synthesis and repair. Moreover, borrowing techniques from [26], we used MaxSAT to identify the set of candidates to repair.

V. EXPERIMENTAL EVALUATION

The goal of experimental evaluation is to showcase the adaptability and performance of the proposed method, SAMYAK¹, across various settings of specifications.

a) *Choice of benchmarks:* Our formulation distinguishes between hard and soft constraints, but no standardized benchmarks exist. We therefore construct synthetic benchmarks where significant output bits are enforced as hard constraints, and the remaining bits as soft. This gradation reflects real-world settings where certain bits dominate correctness—for example, in error correction codes [28], control systems, arithmetic [29], and programmable logic devices [30].

We focus on three key scenarios:

- 1) **Logic/control circuits:** We use the `router.v` benchmark from the EPFL suite, implementing minimal XY-routing. Dimension-order (XY) routing is a simple, deadlock-free, deterministic strategy commonly used in

¹SAMYAK implementation and benchmarks are available: <https://github.com/KushagraGupta02/SAMYAK/>

TABLE I
AREA (IN μm^2) AND ACCURACY OF RETURNED APPROXIMATE CIRCUIT
COMPARISON FOR ROUTER ARRAY.

# Router	Exact Circuit	HS_1		HS_2	
		Area	Accuracy	Area	Accuracy
2	604.458	397.966	1.000	397.970	1.000
3	956.903	662.182	1.000	716.621	1.000
4	1392.882	1180.759	1.000	1183.105	1.000
5	1650.997	1004.302	1.000	960.657	1.000
6	2021.744	1820.884	0.797	1820.884	0.797
7	2376.066	1748.143	0.938	1615.800	0.938
8	2647.791	2176.613	0.938	2068.205	0.938
9	3069.222	1863.121	0.844	1901.604	0.844
10	3310.442	2631.365	0.984	2823.309	0.984
11	3654.439	2876.809	0.875	2957.059	0.875
12	3944.467	2593.352	0.953	2654.361	0.953
13	4449.903	2910.599	0.875	2920.454	0.875
14	4806.571	3338.600	0.969	3410.872	0.969
15	5196.090	3908.800	0.953	3903.168	0.953
16	5171.217	3762.847	0.969	3968.870	0.969

TABLE II
TIME (S), AREA (μm^2), ACCURACY, AND REPAIRS FOR SELECTED
BENCHMARKS USING Manthan2 AND SAMYAK VARIANTS.

Metric	hd-08- 32	hd-02- 64	hd-01- 256	hd-03- 384	hd-05- 512	decomp- 16	factor- 12	decomp- 24
Time Manthan2	3.37	7.52	335.61	1044.60	1961.16	77.87	1626.56	TO
Time HS_3	1.80	6.73	200.81	555.53	622.56	0.70	7.01	1.07
Time HS_4	1.94	7.41	266.01	651.79	1311.97	0.77	5.95	1.14
Time HS_5	2.66	7.91	376.16	934.23	1295.79	0.80	5.92	1.23
Area Manthan2	313.96	743.37	2308.49	4591.63	MO	2664.69	16192.73	TO
Area HS_3	92.45	412.51	1070.47	944.23	3226.91	764.96	502.15	1413.53
Area HS_4	98.55	412.51	1071.88	944.23	3228.78	764.96	502.15	1413.53
Area HS_5	88.23	413.45	1033.87	906.22	3218.93	764.96	502.15	1413.53
Accuracy HS_3	0.9375	0.9688	1.0	0.9375	1.0	0.9688	0.828	0.9688
Accuracy HS_4	0.9531	0.9375	1.0	0.9375	1.0	0.9688	0.828	0.9688
Accuracy HS_5	0.9375	0.9375	1.0	0.9375	1.0	0.9688	0.828	0.9688

2D-mesh NoCs and has been shown to often outperform adaptive routing for typical NoC traffic [18], [19], [21], [22], [31], [32]. To study scalability, we replicate $N = 2 \dots 16$ independent routers, modeling larger mesh networks [18]. Since many workloads (e.g., multimedia, ML) tolerate limited routing imprecision, approximate synthesis of router logic is a practical design point.

In router circuits, decisions are often based on specific bits that determine the routing path, e.g. a single bit may indicate the direction of data flow, partitioning the input space and guiding packet routing. Therefore, we impose the following **hard constraints**: (i) HS_1 : most-significant bit outputs correct for $\lceil N/4 \rceil$ routers, (ii) HS_2 : most-significant bit outputs correct for $\lceil N/2 \rceil$ routers.

- 2) **Bit manipulations**: This includes instances involving bit manipulations, which focuses on performing fast bit-level operations. We use operations from Hacker’s Delight and SyGuS [23], such as finding the rightmost one bit and counting trailing zeros. We evaluate 10 instances with input sizes from 8 to 512 bits, yielding 60 benchmarks (10×6 widths).
- 3) **Arithmetic tasks**: We study non-deterministic problems without unique function definitions [25], namely factorization [24] (synthesizing non-trivial factors Y_1, Y_2 of X) and decomposition [25] (synthesizing $Y_1, Y_2 \neq 0$

such that $X = Y_1 + Y_2$). We generate 6 factorization variants with product widths from 8 to 24 bits, and 6 decomposition variants with sum widths from 8 to 128 bits. Since efficient factorization is computationally hard [33], [34], approximate synthesis with provable guarantees is especially valuable in these scenarios; for instance, even partial recovery of factors—such as certain bits, is informative in cryptographic contexts [34]–[36].

For **Bit manipulation** and **Arithmetic tasks**, we enforce: (i) HS_3 : middle-bit correct, (ii) HS_4 : most-significant bit correct, (iii) HS_5 : top five bits correct. All other outputs are treated as soft constraints.

In total, we evaluate **246 benchmarks**: 15 router arrays $\times 2$ hard constraints plus 72 bit-level/arithmetic instances $\times 3$ hard constraints.

b) Experimental Setup: All our experiments were conducted on a high-performance computer cluster with each node consisting of a E5-2695 v2 CPU with 48 cores and 377GB of RAM, with a memory limit set to 2GB per core. All tools were run in a single-threaded mode on a single core. Satisficing measure is computed as per Equation 1. We set the satisficing threshold to 75%.

c) Tools: Our proposed method benefits from the collective contributions of the constraint solving community. For instance, we used MUSER2 to compute group-minimal UnsatCores [37], and MUST to enumerate multiple UnsatCores [38]. We employed ApproxMC to approximate the model count of a given formula [39], and CMSGen to generate samples for learning an initial system [27]. We also used the python-sat (PySAT) library [40] to obtain satisfying assignments. Furthermore, we relied on Scikit-Learn for learning decision trees used to synthesize candidate functions $F(X)$ [41], [42].

d) Comparison with SOTA: We compared SAMYAK with state-of-the-art functional synthesizer tool Manthan2 [26], [43], which outperforms the state-of-the-art SyGuS tools over bitvector theory [44]. Manthan2 generates provably correct functions on specification $\varphi(X, Y)$ where $\varphi(X, Y)$ is the union of $\varphi_S(X, Y)$ and $\varphi_H(X, Y)$. We considered the timeout of 3600 seconds for both SAMYAK and Manthan2.

Through our experimental evaluation, we aimed to showcase the effectiveness of SAMYAK. In particular, we sought to answer the following questions:

- 1) How *compact* are the synthesized approximate circuits with proven guarantees?² Specifically, what is the reduction in circuit area of the synthesized system, and how does it scale with increasing input size?

²To evaluate area, we convert the synthesized Verilog circuit into a flattened netlist and map it to a target standard cell library e.g., `gsc145nm.lib`, followed by standard optimization passes to obtain final netlist statistics. This flow, aligned with conventional synthesis methodology, enables a quantitative comparison of the compactness and scalability of circuits generated by SAMYAK against other approaches.

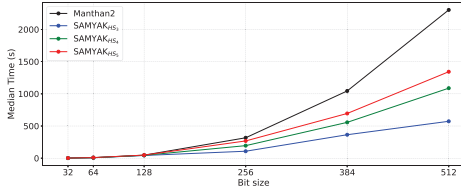


Fig. 2. Comparison of Manthan2 and SAMYAK with different hard constraints for benchmarks over bit-level operations as we increase the bitsize of input instance. Timeout: 3600s.

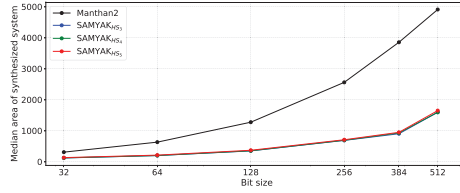


Fig. 3. Comparison of area of system synthesized by Manthan2 and SAMYAK with different hard constraints for benchmarks over bit-level operations as we increase the bitsize of input instance.

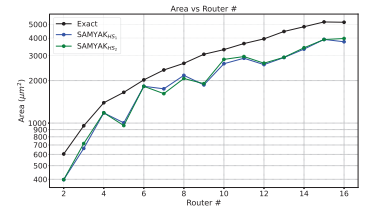


Fig. 4. Comparison of area of system synthesized by SAMYAK and exact circuit with different hard constraints for router array benchmarks.

- 2) How much do we lose when synthesizing approximately correct systems? How *good* is the system synthesized by SAMYAK?
- 3) Is SAMYAK better than the traditional all-or-nothing paradigm? That is, is the performance of SAMYAK comparable to the complete synthesis of the specification involving both hard and soft constraints?

e) Summary of results: In terms of compactness, approximate synthesis with provable guarantees consistently reduced the circuit area across all 246 benchmarks under different hard-constraint configurations, achieving an average area reduction of **26.7%**. Although we set a satisficing threshold of 75%—i.e., accuracy over the entire input space rather than selected test cases—we observed that 88% of benchmarks achieved accuracy above 90%. For bit-manipulation and arithmetic benchmarks, where relation specifications are available, we used Manthan2 to synthesize exact circuits. Compared to Manthan2, our tool SAMYAK achieved at least a $1.5\times$ speedup in **83%** of instances, while reducing area by at least **35%**.

f) Results: Table I compares the exact circuit area against SAMYAK under two different constraint configurations on the router-array benchmarks. Column 2 reports the area of the exact circuit for each router count. Columns 3-4 present the synthesized circuit area and accuracy obtained by SAMYAK under HS_1 and HS_2 (MSB output constraints on $\lceil N/4 \rceil$ and $\lceil N/2 \rceil$ routers, respectively). Each row corresponds to the router array size. Across the 30 (15×2) evaluated benchmarks, SAMYAK achieved area reductions in every instance, with **all** benchmarks saving at least **10%**, **23** above **20%**, and even for 16 routers, a **27%** reduction, demonstrating scalability.

Table II compares Manthan2 and SAMYAK on a representative subset of the bit-level and arithmetic benchmarks³ with each column corresponding to one benchmark. Row 1 shows the synthesis times for Manthan2, which targets full correctness over all output bits. Rows 2-4 report SAMYAK results under hard constraints on the middle bit (HS_3), the MSB (HS_4), and the top five MSBs (HS_5), with soft constraints on remaining bits. Row 5 shows Manthan2’s circuit area, and rows 6-8 show areas for SAMYAK under the respective constraint settings.

³detailed results on each benchmark are available at <https://github.com/KushagraGupta02/SAMYAK>

As shown in Table II, for bit-level operations (hd-*) with bit-widths below 64, both Manthan2 and SAMYAK exhibit comparable performance. However, SAMYAK performs additional tasks, such as computing satisficing measures and verifying hard constraint compliance. As input bit-width increases, Manthan2 slows down significantly compared to SAMYAK, failing to synthesize functions within the time limit for 6 instances. SAMYAK was able to synthesize functions for all except 1 instance. Figures 2, 3 represent the comparisons of synthesis time and synthesized circuit area for Manthan2 and SAMYAK for the bit-level benchmarks. Each point $\langle x, y \rangle$ corresponds to input bit-size x and the median time or area y . Both time and area generally increase with bit-width, but SAMYAK scales further than Manthan2, which fails to synthesize larger inputs (e.g., 512-bit for some bitlevel benchmarks, or beyond 12-16 bits for arithmetic ones). Figure 4 shows comparisons of synthesized circuit area with the exact router array circuit area.

Remark 1: The efficacy of SAMYAK lies in its handling of constraint gradation. As more soft constraints are treated as hard constraints, SAMYAK may take longer to synthesize the required functions. If all constraints are hard, the synthesis time for SAMYAK could be comparable to complete synthesis engines or possibly even longer, attributed to multiple calls on the counter for computing a satisficing measure at each step.

VI. CONCLUSION

Synthesis is a fundamental problem. Our work introduces SAMYAK, a domain-agnostic framework that synthesizes systems satisfying all hard constraints while ensuring the satisficing measure for soft constraints exceeds a predefined threshold. This approach bridges the gap between exact synthesis, which is often too resource-intensive, and traditional approximate synthesis, which sacrifices formal guarantees and evaluates accuracy only on sampled test cases.

Empirically, SAMYAK scales better than traditional approaches, achieves accuracy $\geq 80\%$ across all benchmarks, and reduces area by about 20%, while maintaining provable guarantees over hard constraints. Extending SAMYAK to constraints modulo theories is a promising future direction.

Acknowledgements: Kushagra Gupta and Priyanka Golia were supported by Indian Govt ANRF early career grant (ANRF/ECRG/2024/005777/ENS).

REFERENCES

- [1] G. Fedyukovich and A. Gupta, "Functional synthesis with examples," in *Proc. of CP*, 2019.
- [2] S. Gulwani, O. Polozov, R. Singh *et al.*, "Program synthesis," *Foundations and Trends in Programming Languages*, 2017.
- [3] J. Kim, Q. Hu, L. D'Antoni, and T. Reps, "Semantics-guided synthesis," in *Proc. of POPL*, 2021.
- [4] E. Kitzelmann, "Inductive programming: A survey of program synthesis techniques," in *International workshop on approaches and applications of inductive programming*. Springer, 2009, pp. 50–73.
- [5] V. Kuncak, M. Mayer, R. Piskac, and P. Suter, "Complete functional synthesis," *ACM Sigplan Notices*, 2010.
- [6] J. Huang, J. Lach, and G. Robins, "A methodology for energy-quality tradeoff using imprecise hardware," in *Proc. of DAC*. IEEE, 2012.
- [7] Q. Xu, T. Mytkowicz, and N. S. Kim, "Approximate computing: A survey," *IEEE Design & Test*, 2015.
- [8] I. Scarabottolo, G. Ansaloni, G. A. Constantinides, L. Pozzi, and S. Reda, "Approximate logic synthesis: A survey," *Proceedings of the IEEE*, 2020.
- [9] S. Venkataramani, A. Sabne, V. Kozhikkottu, K. Roy, and A. Raghunathan, "Salsa: systematic logic synthesis of approximate circuits," in *Proceedings of the 49th Annual Design Automation Conference*, ser. DAC '12. New York, NY, USA: Association for Computing Machinery, 2012, pp. 796–801. [Online]. Available: <https://doi.org/10.1145/2228360.2228504>
- [10] J. Castro-Godínez, H. Barrantes-García, M. Shafique, and J. Henkel, "AxlS: A framework for approximate logic synthesis based on netlist transformations," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 68, no. 8, pp. 2845–2849, 2021.
- [11] R. Venkatesan, A. Agarwal, K. Roy, and A. Raghunathan, "Macaco: Modeling and analysis of circuits for approximate computing," in *IEEE/ACM International Conference on Computer-Aided Design, Digest of Technical Papers, ICCAD*, 11 2011, pp. 667–673.
- [12] S. Hashemi, R. I. Bahar, and S. Reda, "Drum: A dynamic range unbiased multiplier for approximate applications," in *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE Press, 2015, pp. 418–425. [Online]. Available: <https://doi.org/10.1109/ICCAD.2015.7372600>
- [13] D. Shin and S. K. Gupta, "Approximate logic synthesis for error tolerant applications," in *Proceedings of the Conference on Design, Automation and Test in Europe*, ser. DATE '10. Leuven, BEL: European Design and Automation Association, 2010, pp. 957–960.
- [14] J. Miao, A. Gerstlauer, and M. Orshansky, "Approximate logic synthesis under general error magnitude and frequency constraints," in *Proceedings of the International Conference on Computer-Aided Design*, ser. ICCAD '13. IEEE Press, 2013, pp. 779–786.
- [15] A. Ranjan, A. Raha, S. Venkataramani, K. Roy, and A. Raghunathan, "Aslan: Synthesis of approximate sequential circuits," in *Proceedings -Design, Automation and Test in Europe, DATE*, 03 2014.
- [16] A. Chandrasekharan, M. Soeken, D. Große, and R. Drechsler, "Approximation-aware rewriting of aigs for error tolerant applications," in *Proceedings of the 35th International Conference on Computer-Aided Design*, ser. ICCAD '16. New York, NY, USA: Association for Computing Machinery, 2016. [Online]. Available: <https://doi.org/10.1145/2966986.2967003>
- [17] S. Hashemi, H. Tann, and S. Reda, "Blasys: approximate logic synthesis using boolean matrix factorization," in *Proceedings of the 55th Annual Design Automation Conference*, ser. DAC '18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: <https://doi.org/10.1145/3195970.3196001>
- [18] B. Edwards, D. Wentzlaff, L. Bao, H. Hoffmann, C.-C. Miao, C. Ramey, M. Mattina, P. Griffin, A. Agarwal, and J. F. Brown III, "On-Chip Interconnection Architecture of the Tile Processor," *IEEE Micro*, vol. 27, no. 05, pp. 15–31, Sep. 2007. [Online]. Available: <https://doi.ieeecomputersociety.org/10.1109/MM.2007.89>
- [19] S. Murali, D. Atienza, L. Benini, and G. De Micheli, "A multi-path routing strategy with guaranteed in-order packet delivery and fault-tolerance for networks on chip," in *2006 43rd ACM/IEEE Design Automation Conference*, 2006, pp. 845–848.
- [20] L. Amaru, P.-E. Gaillardon, and G. De Micheli, "The eplf combinational benchmark suite," in *Proceedings of the 24th International Workshop on Logic & Synthesis (IWLS)*, 2015. [Online]. Available: <https://infoscience.epfl.ch/handle/20.500.14299/113476>
- [21] A. Agarwal and R. Shankar, "Survey of network on chip (noc) architectures and contributions," *Journal of Engineering, Computing and Architecture*, vol. 3, no. 1, pp. 1–15, 2009.
- [22] Dally and Seitz, "Deadlock-free message routing in multiprocessor interconnection networks," *IEEE Transactions on Computers*, vol. C-36, no. 5, pp. 547–553, 1987.
- [23] "SyGuS: syntax-guided synthesis competition 2019," 2019. [Online]. Available: <https://sygus.org/comp/2019/>
- [24] S. Akshay, S. Chakraborty, A. K. John, and S. Shah, "Towards parallel boolean functional synthesis," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2017, pp. 337–353.
- [25] D. Fried, L. M. Tabajara, and M. Y. Vardi, "BDD-based boolean functional synthesis," in *Proc. of CAV*, 2016.
- [26] P. Golia, S. Roy, and K. S. Meel, "Manthan: A data driven approach for boolean function synthesis," in *Proc. of CAV*, 2020.
- [27] P. Golia, M. Soos, S. Chakraborty, and K. S. Meel, "Designing samplers is easy: The boon of testers," in *Proc. of FMCAD*, 2021.
- [28] J. P. Odenwalder, *Error control coding handbook*. Linkabit Corporation San Diego, 1976, vol. 15.
- [29] V. Vijay, M. Sreevani, E. M. Rekha, K. Moses, C. S. Pittala, K. S. Shaik, C. Koteshwaramma, R. J. Sai, and R. R. Vallabhuni, "A review on n-bit ripple-carry adder, carry-select adder and carry-skip adder," *Journal of VLSI circuits and systems*, vol. 4, no. 01, pp. 27–32, 2022.
- [30] E. H. Currie and E. H. Currie, "Programmable logic," *Mixed-Signal Embedded Systems Design: A Hands-on Guide to the Cypress PSoc*, pp. 291–346, 2021.
- [31] W. J. Dally and B. Towles, *Principles and Practices of Interconnection Networks*, 1st ed. Morgan Kaufmann, 2004. [Online]. Available: <https://dl.acm.org/doi/book/10.1016/B978-155860780-9/50005-0>
- [32] F. Moraes, N. Calazans, A. Mello, L. Möller, and L. Ost, "Hermes: an infrastructure for low area overhead packet-switching networks on chip," *Integration*, vol. 38, no. 1, pp. 69–93, 2004. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0167926004000185>
- [33] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Commun. ACM*, vol. 21, no. 2, p. 120–126, Feb. 1978. [Online]. Available: <https://doi.org/10.1145/359340.359342>
- [34] J. Blömer and A. May, "New partial key exposure attacks on rsa," in *Advances in Cryptology - CRYPTO 2003*, D. Boneh, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 27–43.
- [35] D. Boneh *et al.*, "Twenty years of attacks on the rsa cryptosystem," *Notices of the AMS*, vol. 46, no. 2, pp. 203–213, 1999.
- [36] G. D'Alconzo, A. Esser, A. Gangemi, and C. Sanna, "Sneaking up the ranks: Partial key exposure attacks on rank-based schemes," *Cryptology ePrint Archive, Paper 2024/2070*, 2024. [Online]. Available: <https://eprint.iacr.org/2024/2070>
- [37] A. Belov and J. Marques-Silva, "MUSer2 : An efficient mus extractor, system description," 2012.
- [38] J. Bendík and I. Černá, "Must: minimal unsatisfiable subsets enumeration tool," in *Proc. of TACAS*. Springer, 2020.
- [39] M. Soos and K. S. Meel, "Bird: Engineering an efficient CNF-XOR sat solver and its applications to approximate model counting," in *Proc. of the AAAI*, 2019.
- [40] A. Ignatiev, A. Morgado, and J. Marques-Silva, "PySAT: A Python toolkit for prototyping with SAT oracles," in *Proc. of SAT*, 2018.
- [41] "sklearn.tree.decisiontreeclassifier." [Online]. Available: <https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html>
- [42] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg *et al.*, "Scikit-learn: Machine learning in python," *the Journal of machine Learning research*.
- [43] P. Golia, F. Slivovsky, S. Roy, and K. S. Meel, "Engineering an efficient boolean functional synthesis engine," in *Proc. of ICCAD*, 2021.
- [44] P. Golia, S. Roy, and K. S. Meel, "Program synthesis as dependency quantified formula modulo theory," in *Proc. of IJCAI*, 2021.