Manthan: A Data-Driven Approach for Boolean Function Synthesis*

Priyanka Golia[1,2], Subhajit Roy[1], and Kuldeep S. Meel[2]

[1] Computer Science and Engineering, Indian Institute of Technology Kanpur, India
pgolia,subhajit@cse.iitk.ac.in
[2] School of Computing, National University of Singapore, Singapore
meel@comp.nus.edu.sg

Abstract. Boolean functional synthesis is a fundamental problem in computer science with wide-ranging applications and has witnessed a surge of interest resulting in progressively improved techniques over the past decade. Despite intense algorithmic development, a large number of problems remain beyond the reach of the state of the art techniques.
Motivated by the progress in machine learning, we propose Manthan, a novel data-driven approach to Boolean functional synthesis. Manthan views functional synthesis as a classification problem, relying on advances in constrained sampling for data generation, and advances in automated reasoning for a novel proof-guided refinement and provable verification. On an extensive and rigorous evaluation over 609 benchmarks, we demonstrate that Manthan significantly improves upon the current state of the art, solving 356 benchmarks in comparison to 280, which is the most solved by a state of the art technique; thereby, we demonstrate an increase of 76 benchmarks over the current state of the art. Furthermore, Manthan solves 60 benchmarks that none of the current state of the art techniques could solve. The significant performance improvements, along with our detailed analysis, highlights several interesting avenues of future work at the intersection of machine learning, constrained sampling, and automated reasoning.

1 Introduction

Given an existentially quantified Boolean formula $\exists Y F(X,Y)$ over the set of variables X and Y, the problem of Boolean functional synthesis is to compute a vector of Boolean functions, denoted by $\Psi(X) = \langle \psi_1(X), \psi_2(X), \ldots, \psi_{|Y|}(X) \rangle$, and referred to as Skolem function vector, such that $\exists Y F(X,Y) \equiv F(X, \Psi(X))$. In the context of applications, the sets X and Y are viewed as inputs and outputs, and the formula $F(X,Y)$ is viewed as a functional specification capturing the relationship between X and Y, while the Skolem function vector $\Psi(X)$ allows one to determine the value of Y for the given X by evaluating Ψ. The study of Boolean functional synthesis traces back to Boole [12], and over the decades, the problem has found applications in a wide variety of domains such as certified

* The open source tool is available at https://github.com/meelgroup/manthan

QBF solving [8,9,36,41], automated program repair [27], program synthesis [44], and cryptography [35].

Theoretical investigations have demonstrated that there exist instances where Boolean functional synthesis takes super-polynomial time. On the other hand, practical applicability has necessitated the development of algorithms with progressively impressive scaling. The algorithmic progress for Boolean functional synthesis has been driven by a diverse set of techniques: (i) the usage of incremental determinization employing the several heuristics in state-of-the-art Conflict Driven Clause Learning (CDCL) solvers [41], (ii) usage of decomposition techniques employing the progress in knowledge compilation [6,19,28,45], and (iii) Counter-Example Guided Abstraction Refinement (CEGAR)-based techniques relying on usage of SAT solvers as black boxes [4,6,5,28]. While the state of the art techniques are capable of handling problems of complexity beyond the capability of tools a decade ago, the design of scalable algorithms capable of handling industrial problems remains the holy grail.

In this work, we take a step towards the above goal by proposing a novel approach, called Manthan, at the intersection of machine learning, constrained sampling, and automated reasoning. Motivated by the unprecedented advances in machine learning, we view the problem of functional synthesis through the lens of multi-class classification aided by the generation of the data via constrained sampling and employ automated reasoning to certify and refine the learned functions. To this end, the architecture of Manthan comprises of the following three novel techniques:

Data Generation The state of the art machine learning techniques use training data represented as a set of samples where each sample consists of valuations to features and the corresponding label. In our context, we treat X as the features and Y as labels. Unlike the standard setup of machine learning wherein for each assignment to X, there is a unique label, i.e. assignment to Y, the relationship between X and Y is captured by a relation and not necessarily a function. To this end, we design a weighted sampling strategy to generate a *representative* data set that can be fitted using a *compactly sized* classifier. The weighted sampling strategy, implemented using state of the constrained sampler, seeks to uniformly sample input variables (X) while biasing the valuations of output variables towards a particular value.

Dependency-Driven Classifier for Candidates Given training data viewed as a valuation of *features* (X) and their corresponding labels (Y), a natural approach from machine learning perspective would be to perform multi-class classification to obtain $Y = h(X)$, where h is a symbolic representation of the learned classifier. Such an approach, however, can not ensure that h can be expressed as a vector of Boolean functions. To this end, we design a dependency aware classifier to construct a vector of decision trees corresponding to each Y_i, wherein each decision tree is expressed as a Boolean function.

Proof-Guided Refinement Since machine learning techniques often produce good but inexact approximations, we augment our method with automated reasoning techniques to verify the correctness of decision tree-based candi-

date Skolem functions. To this end, we perform a counterexample driven refinement approach for candidate Skolem functions.

To fully utilize the impressive test accuracy attained by machine learning models, we design a *proof-guided refinement* approach that seeks to identify and apply *minor* repairs to the candidate functions, in an iterative manner, until we converge to a provably correct Skolem function vector. In a departure from prior approaches utilizing the Shannon expansion and self-substitution, we first use a MaxSAT solver to determine potential repair candidates, and employ unsatisfiability cores obtained from the infeasibility proofs capturing the reason for current candidate functions to meet the specification, to construct a *good repair*.

Finally, We perform an extensive evaluation over a diverse set of benchmarks with state-of-the-art tools, viz. C2Syn[4], BFSS[5], and CADET[39]. Of 609 benchmarks, Manthan is able to solve 356 benchmarks while C2Syn, BFSS, and CADET solve 206, 247, and 280 benchmarks respectively. Significantly, Manthan can solve 60 benchmarks beyond the reach of all the other existing tools extending the reach of functional synthesis tools. We then perform an extensive empirical evaluation to understand the impact of different design choices on the performance of Manthan. Our study reveals several surprising observations arising from the inter-play of machine learning and automated reasoning.

Manthan owes its runtime performance to recent advances in machine learning, constrained sampling, and automated reasoning. Encouraged by Manthan's scalability, we will seek to extend the above approach to related problem domains such as automated program synthesis, program repair, and reactive synthesis.

The rest of the paper is organized as follows: We first introduce notations and preliminaries in Section 2. We then discuss the related work in Section 3. In Section 4 we present an overview of Manthan and give an algorithmic description in Section 5. We then describe the experimental methodology and discuss results in Section 6. Finally, we conclude in Section 7.

2 Notations and Preliminaries

We use lower case letters (with subscripts) to denote propositional variables and upper case letters to denote a subset of variables. The formula $\exists Y F(X,Y)$ is existentially quantified in Y, where $X = \{x_1, \cdots, x_n\}$ and $Y = \{y_1, \cdots, y_m\}$. For notational clarity, we use F to refer to $F(X,Y)$ when clear from the context. We denote $Vars(F)$ as the set of variables appearing in $F(X,Y)$. A literal is a boolean variable or its negation. We often abbreviate universally (resp. existentially) quantified variables as universal (resp. existential) variables.

A *satisfying assignment* of a formula $F(X,Y)$ is a mapping $\sigma : Vars(F) \to \{0,1\}$, on which the formula evaluates to True. For $V \subseteq Vars(F)$, $\sigma[V]$ represents the truth values of variables in V in a satisfying assignment σ of F. We denote the set of all witnesses of F as R_F. For a formula in conjunctive normal form, the *unsatisfiable core*(UnsatCore) is a subset of clauses of the formula for which no satisfying assignment exists.

We use $F(X,Y)|_{y_i=b}$ to denote *substitutions*: a formula obtained after substituting every occurrence of y_i in $F(X,Y)$ by b, where b can be a constant (0 or 1) or a formula. The operator *ite(condition,exp1,exp2)* is used to represent the if-else case: if the *condition* is true, then it returns *exp1*, else it returns *exp2*.

A variable y_i is considered as a *positive unate* if and only if $F(X,Y)|_{y_i=0} \wedge \neg F(X,Y)|_{y_i=1}$ is UNSAT and a *negative unate* if and only if $F(X,Y)|_{y_i=1} \wedge \neg F(X,Y)|_{y_i=0}$ is UNSAT [5].

Given a function vector $\langle \psi_1, \ldots, \psi_m \rangle$ for the vector of variables $\langle y_1, \ldots y_m \rangle$ such that ψ_i is the function corresponding to y_i, we say that there exists a partial order $\prec_d$ over the variables $\{y_1, \ldots y_m\}$ such that $y_i \prec_d y_j$ if ψ_i depends on y_j.

In decision tree learning, a fraction of incorrectly assigned labels refer to the *impurity*. We use Gini Index [38] as a measure of *impurity* for a class label. The *impurity decrease* at a node is the difference of its impurity to the mean of impurities of its children. The *minimum impurity decrease* is a hyper-parameter used to control the maximum allowable impurity at the leaf nodes, thereby providing a lever for how closely the classifier fits the training data.

Given a propositional formula $F(X,Y)$ and a weight function $W(\cdot)$ assigning non-negative weights to every literal, we refer to the *weight* of a satisfying assignment σ, denoted as $W(\sigma)$, as the product of weights of all the literals appearing in σ, i.e., $W(\sigma) = \prod_{l \in \sigma} W(l)$. A *sampler* $\mathcal{A}(\cdot,\cdot)$ is a probabilistic generator that guarantees $\forall \sigma \in R_F$, $\mathsf{Pr}\left[\mathcal{A}(F, \mathsf{Bias}) = \sigma\right] \propto W(\sigma)$.

We use a function Bias that takes a mapping from a sequence of variables to the desired weights of their positive literals, and assigns corresponding weights to each of the positive literals. We use a simpler notation, `Bias(a,b)` to denote that positive literals corresponding to all universal variables are assigned a weight `a` and positive literals corresponding to all existential variables are assigned a weight `b`. For example, $\mathsf{Bias}(0.5,\ 0.9)$ assigns a weight of 0.5 to the positive literals of the universally quantified variables and 0.9 to the positive literals of the existentially quantified variables.

Problem Statement: Given a Boolean specification $F(X,Y)$ between set of inputs $X = \{x_1, \cdots, x_n\}$ and vector of outputs $Y = \langle y_1, \cdots, y_m \rangle$, the problem of *Skolem function synthesis* is to synthesize a function vector $\Psi = \langle \psi_1(X), \cdots, \psi_m(X) \rangle$ such that $y_i \leftrightarrow \psi_i(X)$ and $\exists Y F(X,Y) \equiv F(X,\Psi)$. We refer to Ψ as the *Skolem function vector* and ψ_i as the *Skolem function* for y_i.

A variable y_i is called self-substituted variable, if the Skolem function ψ_i corresponding to y_i is set to $F(X,Y)|_{y_i=1}$ [19].

Given a formula $\exists Y F(X,Y)$ and a Skolem function vector Ψ, we refer to $E(X,Y,Y')$ as an *error formula* [28], where $Y' = \{y'_1, \cdots, y'_{|Y|}\}$, and $Y' \neq Y$.

$$E(X,Y,Y') = F(X,Y) \wedge \neg F(X,Y') \wedge (Y' \leftrightarrow \Psi) \tag{1}$$

We use the following theorems from prior work:

Theorem 1 ([28]). *Ψ is a Skolem function if and only if $E(X,Y,Y')$ is UNSAT.*

Theorem 2 ([5]). *If y_i is positive(resp negative) unate in $F(X,Y)$, then $\psi_i = 1$ (resp $\psi_i = 0$) is the Skolem function for y_i.*

3 Related Work

The origins of the problem of Boolean functional synthesis traces back to Boole's seminal work [12], which was subsequently rigorously pursued, albeit focused on decidability, by Lowenheim and Skolem [33]. The complexity theoretic studies have shown that there exist instances where Boolean functional synthesis takes super polynomial time and was also shown that there exist instances for which polynomial size Skolem function vector does not suffice unless Polynomial Hierarchy (PH) collapses [5].

Motivated by the success of the CEGAR (Counter-Example Guided Abstraction Refinement) approach in model checking, CEGAR-based approaches have been pursued in the context of synthesis as well, where the key idea is to use a Conflict-Driven Clause Learning (CDCL) SAT solver to verify and refine the candidate Skolem functions [4,5,6,28].

Another line of work has focused on the representation of specification, i.e., $F(X,Y)$, in representations that are amenable to efficient synthesis for a class of functions. The early approaches focused on ROBDD representation building on the functional composition approach proposed by Balabanov and Jiang [8]. Building on Tabajara and Vardi's ROBDD-based approach [45], Chakraborty et al. extended the approach to factored specifications [14]. It is worth mentioning that factored specifications had earlier been pursued in the context of CEGAR-based approaches. Motivated by the success of knowledge compilation in the field of probabilistic reasoning, Akshay et al. achieved a significant breakthrough over a series of papers [5,6,28] to propose a new negation normal form, SynNNF [4]. The generalization and a functional specification presented in SynNNF is amenable to efficient functional synthesis [4]. Another line of work focused on the usage of *incremental determinization* to incrementally construct the Skolem functions [25,30,36,39,41].

Several approaches have been proposed for the particular case when the specification, $\exists Y F(X,Y)$ is valid, i.e., $\forall X \exists Y F(X,Y)$ is True. Inspired by the sequential relational decomposition, Chakraborty et al. [14] recently proposed an approach focused on viewing each CNF clause of the specification consisting of *input and output* clauses and employing a *cooperation*-based strategy. The progress in modern CDCL solvers has led to an exploration of usage of heuristics for problems in complexity classes beyond NP. This has led to work on the extraction of Skolem functions from the proofs constructed for the formulas expressed as $\forall X \exists Y F(X,Y)$ [8,9].

The performance of Manthan crucially depends on its ability to employ constrained sampling, which has witnessed a surge of interest with approaches ranging from those based on hashing-based techniques [15], knowledge compilation [24,42], augmentation of SAT solvers with heuristics [43].

The recent success of machine learning has led to several attempts to the usage of machine learning in several related synthesis domains such as program synthesis [7], invariant generation, decision-tree for functions in Linear Integer Arithmetic theory using pre-specified examples [18], strategy synthesis for QBF [26]. Use of data-driven approaches for invariant synthesis has been investigated in the ICE learning framework [17,20,21] aimed with data about the program behavior from test executions, it proposes invariants by learning from data, checks for inductiveness and, on failure, extend the data by the generated counterexamples. The usage of proof-artifacts such as unsat cores has been explored in verification since early 2000s [23] and in program repair in Wolverine [46], while MaxSAT has been used in program debugging in [10,29].

4 Manthan: An overview

In this section, we provide an overview of our proposed framework, Manthan, before divulging into core algorithmic details in the following section. Manthan takes in a function specification, represented as $F(X, Y)$, and returns a Skolem function vector $\Psi(X)$ such that $\exists Y F(X, Y) \equiv F(X, \Psi(X))$. As shown in Figure 1 Manthan consists of following three phases:

1. Preprocess employs state-of-the-art pre-processing techniques on F to compute a partial Skolem function vector.
2. LearnSkF takes in the pre-processed formula and uses constrained samplers, and classification techniques to compute candidate Skolem functions for all the variables in Y.
3. Refine performs verification and proof-guided refinement procedure wherein a SAT solver is employed to verify the correctness of candidate functions and a MaxSAT solver in conjunction with a SAT solver is employed to refine the candidate functions until the entire candidate Skolem function vector passes the verification check.

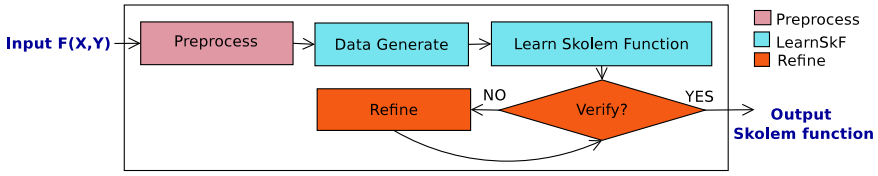


Fig. 1: Overview of Manthan

We now provide a high-level description of different phases to highlight the technical challenges, which provides context for several algorithmic design choices presented in the next section.

4.1 Phase 1: Preprocess

Preprocess focuses on pre-processing of the formula to search for unates among the variables in Y; if y_i is positive (resp. negative) unate, then $\psi_i = 1$(resp. 0)

suffices as a Skolem function. We employ the algorithmic routine proposed by Akshay et al. [5] to drive this preprocessing.

4.2 Phase 2: LearnSkF

LearnSkF views the problem of functional synthesis through the lens of machine learning where the learned machine learning model for classification of a variable y_i can be viewed as a candidate Skolem function for y_i. We gather training data about the function's behavior by exploiting the progress in constrained sampling to sample solutions of $F(X,Y)$. Recall that $F(X,Y)$ defines a relation (and not necessarily a function) between X and Y, and the machine learning techniques typically assume the existence of function between features and labels, necessitating the need for sophisticated sampling strategy as discussed below. Moving on to features and labels, since we want to learn Y in terms of X, we view X as a set of features while assignments to Y as a set of class labels.

The off-the-shelf classification techniques typically require that the size of training data is several times larger than the size of possible class labels, which would be prohibitively large for the typical problems involving more than thousand variables. To mitigate the requirement of large training data, we make note of two well-known observations in functional synthesis literature: (1) the Skolem function ψ_i for a variable y_i typically does not depend on all the variables in X, (2) A Skolem function vector Ψ where ψ_i depends on variable y_j is a valid vector if the Skolem function ψ_j is not dependent on y_i (i.e., acyclic dependency), i.e., there exists a partial order $\prec_d$ over $\{y_1, \ldots y_m\}$.

The above observations lead us to design an algorithmic procedure where we learn candidate Skolem functions as decision trees in an iterative manner, i.e., one y_i at a time, thereby allowing us to constrain ourselves to the binary classification. The learned classifier can then be represented as the disjunction of all the paths from the root to the leaves in the learnt decision tree. We update the set of possible features for a given y_i depending on the candidate functions generated so far, i.e., valuation of X variables and Y variables, which are not dependent on y_i. Finally, we compute the candidate Skolem function for y_i as the disjunction of labels along edges for all the paths from the root to leaf nodes with label 1. Once, we have the candidate Skolem function vector Ψ, we obtain a valid linear extension, *TotalOrder*, of the partial order $\prec_d$ in accordance to Ψ.

Before moving on to the next phase, we return to the formulation of sampling. The past few years have witnessed the design of uniform [15,42], and weighted samplers [24], and one wonders what kind of sampler should we choose to generate samples for training data. A straightforward choice would be to perform uniform sampling over X and Y, but the relational nature of specification, F, between X and Y offers interesting challenges and opportunities. Recall while F specifies a relation between X and Y, we are interested in a Skolem function, and we would like to tailor our sampling subroutines to allow discovery of Skolem functions with *small* description given the relationship between description and sample complexity. To this end, consider $X = \{x_1, x_2\}$ and $Y = \{y_1\}$, and let $F := (x_1 \vee x_2 \vee y_1)$. Note that F has 7 solutions over $X \cup Y$, out of which $y_1 = 0$

appears in 3 solutions while $y_1 = 1$ appears in 4. Also, note that there are several possible Skolem functions such as $y_1 = \neg(x_1 \wedge x_2)$. Now, if we uniformly sample solutions of F over x_1, x_2, y_1, i.e. Bias(0.5, 0.5), we would see (almost) equal number of samples with $y_1 = 0$ and $y_1 = 1$. A closer look at F reveals that it is possible to construct a Skolem function by knowing that the only case where y_1 cannot be assigned 0 is when $x_1 = x_2 = 0$. To encode this intuition, we propose a novel idea of collecting samples with weighted sampling, i.e., Bias(0.5, q) where q is chosen in a multi-step process of first drawing a small set of samples with both $q = 0.9$ and $q = 0.1$, and then drawing rest of the samples by fixing the value of q following analysis of an initial set of samples. To the best of our knowledge, this is the first application of weighted sampling in the context of synthesis, and our experimental results point to several interesting avenues of future work.

4.3 Phase 3: Refine

The candidate Skolem functions generated in LearnSkF may not always be the actual Skolem functions. Hence, we require a *verification* check to see if candidate Skolem functions are indeed correct; if not, the generated counterexample can be used to *repair* it. The verification query constructs an *error formula* $E(X, Y, Y')$(Formula 1): if unsatisfiable, the candidate Skolem function vector is indeed a Skolem function vector and the procedure can terminate; else, when $E(X, Y, Y')$ is SAT, the solution of $E(X, Y, Y')$ is used to identify and refine the erring functions among the candidate Skolem function vector.

In contrast to prior techniques that apply Shannon expansion or self-substitution, the refinement strategy in Manthan is guided by the view that the candidate function vector from the LearnSkF phase is *almost correct*, and hence, attempts to identify and apply a series of *minor* repairs to the erring functions to arrive at the correct Skolem function vector. To this end, Manthan uses two key techniques: *fault localization* and *repair synthesis*. Let us assume that σ is a satisfying assignment of $E(X, Y, Y')$ and referred to as counterexample for the current candidate Skolem function vector Ψ.

Fault Localization In order to identify the initial candidates to repair for the counterexample σ, Manthan attempts to identify a small number of Skolem functions (correspondingly Y variables) whose outputs must undergo a change for the formula to behave correctly on σ; in other words, it makes a best-effort attempt to ensure that most of the Skolem functions (correspondingly Y variables) can retain their current output on σ while satisfying the formula. Manthan encodes this problem as a partial MaxSAT query with $F(X, Y) \wedge (X \leftrightarrow \sigma[X])$ as a hard constraint and $(Y \leftrightarrow \sigma[Y'])$ as soft constraints. All Y variables whose valuation constraint $(Y \leftrightarrow \sigma[Y'])$ does not hold in the MaxSAT solution are identified as erring Skolem functions that may need to be repaired.

Repair Synthesis Let y_k be the variable corresponding to the erring function, ψ_k, identified in the previous step. To synthesize a repair for the function, Manthan applies a proof-guided strategy: it constructs a formula $G_k(X, Y)$, such

that if $G_k(X,Y)$ is unsatisfiable then ψ_k must undergo a change. The UnsatCore of $G_k(X,Y)$ provides a *reason* that explains the discrepancy between the specification and the current Skolem function.

$$G_k(X,Y) = (y_k \leftrightarrow \sigma[y'_k]) \wedge F(X,Y) \wedge (X \leftrightarrow \sigma[X]) \wedge (\hat{Y} \leftrightarrow \sigma[\hat{Y}])$$
$$\text{where } \hat{Y} \subset Y \text{ and } \hat{Y} = \{\mathit{TotalOrder}[\mathit{index}(y_k)+1], \cdots, \mathit{TotalOrder}[|Y|]\} \quad (2)$$

Manthan uses the UnsatCore to constructs a *repair formula*, say β, as a conjunction over literals in the unsatisfiable core; if ψ_k is *true* with the current valuation of X and $\hat{Y}$, Manthan updates the function ψ_k by conjoining it with the negation of repair formula ($\psi_k \leftarrow \psi_k \wedge \neg\beta$); otherwise, Manthan updates the function ψ_k, by disjoining it with the repair formula ($\psi_k \leftarrow \psi_k \vee \beta$).

Self-substitution for poorly learnt functions Some Skolem functions are difficult to learn through data. In our implementation, the corresponding variables escape the LearnSkF phase with poor candidate functions, thereby requiring a long sequence of incremental repairs for convergence. To handle such scenarios, we make the following observation: though synthesizing Skolem functions via self-substitution[19] can lead to an exponential blowup in the worst case, it is inexpensive if the number of variables synthesized via this technique is small. We use this observation to quickly synthesize a Skolem function for an erring variable if we detect its candidate function is poor (detected by comparing the number of times it enters refinement against an empirically determined threshold). Of course, this heuristic does not scale well if the number of such variables is large; in our experiments, we found less than 20% of the instances solved required self-substitution, and for over 75% of these instances, only one variable needed self-substitution. We elaborate more on the empirical evidence on the success of this heuristic in section 6. A theoretical understanding of the learnability of Boolean functions from data seems to be an interesting direction for future work.

5 Manthan: Algorithmic Description

In this section, we present a detailed algorithmic description of Manthan, whose pseudocode is presented in Algorithm 1. Manthan takes in a formula $F(X,Y)$ as input and returns a Skolem vector Ψ. The algorithm starts off by preprocessing (line 1) the formula $F(X,Y)$ to get the unates (U) and their corresponding Skolem functions (Ψ). Next, it invokes the sampler (line 2) to collect a set of samples(Σ) as training data for the learning phase.

For each of the existential variables that are not unates, Manthan attempts to learn candidate Skolem functions (lines 4-5). To generate a variable order, CandidateSkF uses a collection of sets $d_1, \cdots, d_{|Y|} \in D$, such that $y_i \in d_j$ indicates that y_j depends on y_i. Next, the FindOrder routine (line 6) construct *TotalOrder* of the Y variables in accordance to the dependencies in D. The

Algorithm 1: Manthan($F(X,Y)$)

```
1  Ψ, U ← Preprocess(F(X,Y))
2  Σ ← GetSamples(F(X,Y))
3  D ← ∅
4  foreach y_j ∈ Y \ U do
5  |  ψ_j, D ← CandidateSkF(Σ, F(X,Y), y_j, D)
6  TotalOrder ← FindOrder(D)
7  repeat
8  |  E(X,Y,Y') ← F(X,Y) ∧ ¬F(X,Y') ∧ (Y' ↔ Ψ)
9  |  ret, σ ← CheckSat(E(X,Y,Y'))
10 |  if ret = SAT then
11 |  |  Ψ ← RefineSkF(F(X,Y), Ψ, σ, TotalOrder)
12 until ret = UNSAT
13 Ψ ← Substitute(F(X,Y), Ψ, TotalOrder)
14 return Ψ
```

verification and refinement phase (line 8) commences by constructing the error formula and launching the verification check (line 9). If the error formula is satisfiable, the counterexample model (σ) is used to refine the formula. Once the verification check is successful, the refinement phase ends and the subroutine Substitute is invoked to recursively substitute all $y_i \in Y$ appearing in Skolem functions with their corresponding Skolem functions such that only X variables entirely describe all Skolem functions. The strict variable ordering enforced above ensures that Substitute always succeeds and does not get stuck in a cycle. Finally, the Skolem function vector Ψ is returned.

It is worth noting that Manthan can successfully solve an instance without having to necessarily execute all the phases. In particular, if $U = Y$, then Manthan terminates after Preprocess (i.e., line 1). Similarly, if the CheckSat return UNSAT during the first iteration of loop (lines 8–11), then Manthan does not invoke RefineSkF.

We now discuss each subroutine in detail. The pseudocode for Preprocess, GetSamples and Substitute is deferred to technical report [22].

Preprocess*:* We perform the pre-processing step as described in [5], which performs SAT queries on the formulas constructed as specified in Theorem 2.

GetSamples*:* GetSamples takes $F(X,Y)$ as input and returns a subset of satisfying assignments of $F(X,Y)$. GetSamples first generates a small set of samples (500) with Bias$(0.5, 0.9)$ and calculates m_i for all y_i, m_i is a ratio of number of samples with y_i being 1 to the total number of samples. Similarity, GetSamples generates 500 samples with Bias$(0.5, 0.1)$ and calculates n_i for all y_i, n_i is a ratio of number of samples with y_i being 0 to the total number of samples. Finally, GetSamples generates required number of samples with Bias$(0.5, q)$; for a y_i, q is m_i if both m_i and n_i are in range 0.35 to 0.65, else q is 0.9.

CandidateSkF*:* CandidateSkF, presented in Algorithm 2, assumes access to following three subroutines:

Algorithm 2: CandidateSkF$(\Sigma, F(X,Y), y_j, D)$

```
1  featset ← X
2  foreach y_k ∈ Y \ y_j  do
3  |  if y_j ∉ d_k then
4  |  |  featset ← featset ∪ y_k /* if y_k is not dependent on y_j  */
5  feat, lbl ← Σ_{↓featset}, Σ_{↓y_j}
6  t ← CreateDecisionTree(feat, lbl)
7  foreach  n ∈ LeafNodes(t) do
8  |  if Label(n) = 1  then
9  |  |  π ← Path(t, root, n)
10 |  |  ψ_j ← ψ_j ∨ π
11 foreach y_k ∈ ψ_j do
12 |  d_j ← d_j ∪ y_k ∪ d_k
13 return ψ_j, D
```

1. CreateDecisionTree takes the feature and label sets as input (training data) and returns a decision tree t. We use the ID3 algorithm [38] to construct a decision tree t where the internal node of t represents a feature on which a decision is made, the branches represent partitioning of the training data on the decision, and the leaf nodes represent the classification outcomes (i.e class labels). The ID3 algorithm iterates over the training data, and in each iteration, it selects a new attribute to extend the tree by a new decision node: the selected attribute is one that causes the maximum drop in the impurity of the resulting classes; we use Gini Index [38] as the measure of impurity. The algorithm, then, extends the tree by the selected decision and continues extending building the tree. The algorithm terminates on a path if either it exhausts all attributes for decisions, or the impurity of the resulting classes drop below a (user-specified) impurity decrease parameter.
2. Label takes a leaf node of the decision tree as input and returns the class label corresponding to the node.
3. Path takes a tree t and two nodes of t (node a and node b) as input and outputs a conjunction of literals in the path from node a to node b in t.

As we seek to learn Boolean functions, we employ binary classifiers with class labels `0` and `1`. CandidateSkF shows our algorithm for extracting a Boolean function from the decion trees: lines 2-4 find a feature set (*featset*) to predict y_j. The feature set includes all X variables and the subset of Y variables that are not dependent on y_j. CandidateSkF creates decision tree t using samples Σ over the feature set. Lines 7-10 generate candidate Skolem function ψ_j by iterating over all the leaf nodes of t. In particular, if a leaf node is labeled with `1`, the candidate function is updated by disjoining with the formula returned by subroutine Path. CandidateSkF also updates d_j in D, d_j is set of all Y variables on which, y_j depends. If y_j depends on y_k, then by transitivity y_j also depends on d_k; in line 12, CandidateSkF updates d_j accordingly.

FindOrder*:* FindOrder takes D as an input to output a valid linear extension of the partial order $\prec_d$ defined over $\{y_1, \dots y_m\}$ with respect to the candidate Skolem function vector Ψ.

Algorithm 3: RefineSkF$(F(X,Y), \Psi, \sigma, \mathit{TotalOrder})$

```
1  H ← F(X,Y) ∧ (X ↔ σ[X]); S ← (Y ↔ σ[Y'])
2  Ind ← MaxSATList(H, S)
3  foreach y_k ∈ Ind do
4  |  Ŷ ← {TotalOrder[index(y_k) + 1], ··· , TotalOrder[|Y|]}
5  |  if CheckSubstitute(y_k) then
6  |  |  ψ_k ← DoSelfSubstitution(F(X,Y), y_k, Y \ Ŷ)
7  |  else
8  |  |  G_k ← (y_k ↔ σ[y'_k]) ∧ F(X,Y) ∧ (X ↔ σ[X]) ∧ (Ŷ ↔ σ[Ŷ])
9  |  |  ret, ρ ← CheckSat(G_k)
10 |  |  if ret = UNSAT then
11 |  |  |  C ← FindCore(G_k)
12 |  |  |  β ← ⋀_{l∈C} ite((σ[l] = 1), l, ¬l)
13 |  |  |  ψ_k ← ite((σ[y'_k] = 1), ψ_k ∧ ¬β, ψ_k ∨ β)
14 |  |  else
15 |  |  |  foreach y_t ∈ Y \ Ŷ do
16 |  |  |  |  if ρ[y_t] ≠ σ[y'_t] then
17 |  |  |  |  |  Ind ← Ind.Append(y_t)
18 |  |  |  σ[y_k] ← σ[y'_k]
19 return Ψ
```

RefineSkF*:* RefineSkF is invoked with a counterexample σ. RefineSkF first performs *fault localization* to find the initial set of erring candidate functions; to this end, it calls the MaxSATList subroutine (line 2) with $F(X,Y) \wedge (X \leftrightarrow \sigma[X])$ as hard-constraints and $(Y \leftrightarrow \sigma[Y])$ as soft-constraints. MaxSATList employs a MaxSAT solver to find the solution that satisfies all the hard constraints and maximizes the number of satisfied soft constraints, and then returns a list (Ind) of Y variables such that for each of the variables appearing in (Ind) the corresponding soft-constraint was not satisfied by the optimal solution returned by MaxSAT solver.

Since candidate Skolem function corresponding to the variables in Ind needs to refine, RefineSkF now attempts to synthesize a repair for each of these candidate Skolem functions. Repair synthesis loop (lines 3–19) starts off by collecting the set of Y variables, $\hat{Y}$, on which y_k of Ind can depend on as per the ordering constraints (line 4). Next, it invokes the subroutine CheckSubstitute, which returns True if the candidate function corresponding to y_k has been refined more than a chosen threshold times (fixed to 10 in our implementation), and the corresponding decision tree constructed during execution CandidateSkF has exactly one node. If CheckSubstitute returns true, RefineSkF calls DoSelfSubstitution to perform self-substitution. DoSelfSubstitution takes a formula $F(X,Y)$, an existentially quantified variable y_k and a list of variables which depends on y_k and performs self substitution of y_k with constant 1 in the formula $F(X,Y)$[28].

If CheckSubstitute returns false, RefineSkF attempts a proof-guided repair for y_k. RefineSkF calls CheckSat in line 9 on G_k, which corresponds to formula 2: if G_k is SAT, then CheckSat returns a satisfying assignment(ρ) of G_k in σ, else CheckSat returns unsatisfiable in the result, ret.

1. If ret is UNSAT, we proceed to refine ψ_k such that for $\psi_k(X \mapsto \sigma[X], \hat{Y} \mapsto \sigma[\hat{Y}]) = \sigma[y_k]$. Ideally, we would like to apply a refinement that generalizes to potentially other counter-examples, i.e. solutions of $E(X, Y, Y')$. To this end, RefineSkF calls FindCore with G_k; FindCore returns the list of variables (C) that occur in the clauses of UnsatCore of G_k. Accordingly, the algorithm constructs a *repair formula* β as a conjunction of literals in σ corresponding to variables in C (line 12). If $\sigma[y'_k]$ is 1, then ψ_k is ψ_k with conjunction of negation of β and if $\sigma[y'_k]$ is 0, then ψ_k is ψ_k with disjunction of β.
2. If ret is SAT and ρ is a satisfying assignment of G_k, then there exists a Skolem function vector such that the value of ψ_k agrees with $\sigma[y_k]$ for the valuation of X and $\hat{Y}$ set to $\sigma[X]$ and $\sigma[\hat{Y}]$. However, for any $y_t \in Y \setminus \hat{Y}$ if $\sigma[y'_t] \neq \rho[y'_t]$, then for such a y_t, the Skolem function corresponding to y_t may need to refine . Therefore, RefineSkF adds y_t to list of candidates to refine, Ind. Note that since $\sigma \models E(X, Y, Y')$, there exists at least one iteration of the loop (lines 3– 18) where ret is UNSAT.

Substitute*:* To return the Skolem functions in terms of only X, Manthan invokes Substitute subroutine. For each y_j of Y variable, Substitute consider Y variables that occurs later in *TotalOrder* as $\hat{Y}$. Then, for each y_i of $\hat{Y}$; it substitutes corresponding Skolem function ψ_i in the Skolem function ψ_j of y_j.

An example to illustrate our algorithm is deferred to the technical report [22].

6 Experimental Results

We evaluate the performance of Manthan on the union of all the benchmarks employed in the most recent works [4,5],which includes 609 benchmarks from different sources: Prenex-2QBF track of QBFEval-17[2], QBFEval-18[3], disjunctive[6], arithmetic[45] and factorization[6]. We ran all the tools as per the specification laid out by their authors. We used Open-WBO [34] for our MaxSAT queries and PicoSAT [11] to compute UnsatCore. We used PicoSAT for its ease of usage and we expect further performance improvements by upgrading to one of the state of the art SAT solvers. We have used the Scikit-Learn[37] to create decision trees in LearnSkF phase of Manthan. We have also used ABC [31] to represent and manipulate Boolean functions. To allow for the input formats supported by the different tools, we use the utility scripts available with the BFSS distribution [5] to convert each of the instances to both QDIMACS and Verilog formats. For Manthan, unless otherwise specified, we set the number of samples according to heuristic based on $|Y|$ as described in Section 6.3 and minimum impurity decrease to 0.005. All our experiments were conducted on a high-performance computer cluster with each node consisting of a E5-2690 v3 CPU with 24 cores

and 96GB of RAM, with a memory limit set to 4GB per core. All tools were run in a single-threaded mode on a single core with a timeout of 7200 seconds.

The objective of our experimental evaluation was two-fold: to understand the impact of various design choices on the runtime performance of Manthan and to perform an extensive comparison of runtime performance vis-a-vis state of the art synthesis tools. In particular, we sought to answer the following questions:

1. How does the performance of Manthan compare with state of the functional synthesis engines?
2. How do the usage of different sampling schemes and the quality of samplers impact the performance of Manthan?
3. What is the impact of LearnSkF on the performance of Manthan?
4. What is the distribution of the time spent in each phase of Manthan?
5. How does using MaxSAT solver to identify the potential erring Skolem functions impacts on the performance of Manthan?
6. How does employing self-substitution for some Skolem functions impact Manthan?

We observe that Manthan significantly improves upon state of the art, and solves 356 benchmarks while the state of the art tool can only solve 280; in particular, Manthan solves 60 more benchmarks that could not be solved by any of the state of the art tools. To put the runtime performance statistics in a broader context, the number of benchmarks solved by techniques developed over the past five years range from 206 to 280, i.e., a difference of 74, which is same as an increase of 76 (i.e., from 280 to 356) due to Manthan.

Our experimental evaluation leads to interesting conclusions and several directions for future work. We observe that the performance of Manthan is sensitive to different sampling schemes and the underlying samplers; in fact, we found that biased sampling yields better results than uniform sampling. This raises interesting questions on the possibility of designing specialized samplers for this task. Similarly, we observe interesting trade offs between the number of samples and the minimum impurity decrease in LearnSkF. The diversity of our extensive benchmark suite produces a nuanced picture with respect to time distribution across different phases, highlighting the critical nature of each of the phases to the performance of Manthan. Manthan shows significant performance improvement by using MaxSAT solver to identify candidates to refine. Manthan also has significant performance improvement with self substitution in terms of the required number of refinements.

6.1 Comparison with other tools

We now present performance comparison of Manthan with the current state of the art synthesis tools, BFSS [5], C2Syn [4], BaFSyn [14] and the current state of the art 2-QBF solvers CADET [39],CAQE [40] and DepQBF [32]. The certifying 2-QBF solver produces QBF certificates, that can be used to extract Skolem functions [8]. Developers of BaFSyn and DepQBF confirmed that the

tools produce Skolem function for only valid instances, i.e. when $\forall X \exists Y F(X, Y)$ is valid. Note that the current version of CAQE does not support certification and we have used CAQE version 2 for the experiments after consultation with the developers of CAQE.

Table 1: No. of benchmarks solved by different tools

Total	BaFSyn	CAQE	DepQBF	C2Syn	BFSS	CADET	Manthan	All Tools
609	13	54	59	206	247	280	**356**	476

We present the number of instances solved Table 1. Out of 609 benchmarks, the most number of instances solved by any of the remaining techniques is 280 while Manthan is able to solve 356 instances – a significant improvement over state of the art. We will focus on top 4 synthesis tools from Table 1 for further analysis.

For a deeper analysis of runtime behavior, we present the cactus plot in Figure 2: the number of instances are shown on the x-axis and the time taken on the y-axis; a point (x, y) implies that a solver took less than or equal to y seconds to find Skolem function of x instances on a total of 609 instances. An interesting behavior predicted by cactus plot and verified upon closer analysis is that for instances that can be solved by most of the tools, the initial overhead due to a multi-phase approach may lead to relatively larger runtime for Manthan. However, with the rise in empirically observed hardness of instances, one can observe the strengths of the multi-phase approach. Overall, Manthan solves 76 more instances than the rest of the remaining techniques.

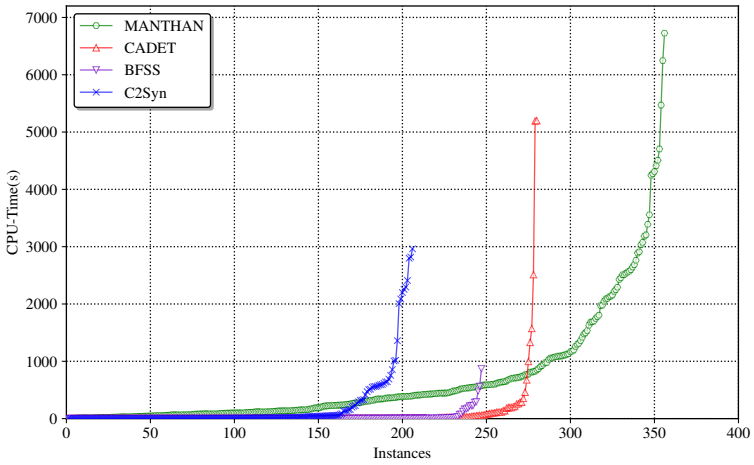


Fig. 2: Manthan versus competing tools for Skolem function synthesis

We show a pairwise comparison of Manthan vis-a-vis other techniques in Table 2. The second row of the table lists the number of instances that were solved by the technique in the corresponding column but not by Manthan while the third row lists the number of instances that were solved by Manthan but not

Table 2: Manthan vs other state-of-the-art tools

		C2Syn	BFSS	CADET	All Tools
Manthan	Less	13	85	111	122
	More	**163**	**194**	**187**	**60**

the corresponding technique. First, we observe that Manthan solves 163, 194, and 187 instances that are not solved by C2Syn, BFSS, and CADET respectively. Though BFSS and CADET solve more than 80 instances that Manthan does not solve, they are not complementary; there are only 121 instances that can be solved by either BFSS or CADET but Manthan fails to solve. A closer analysis of Manthan's performance on these instances revealed that the decision trees generated by CandidateSkF were shallow, which is usually a sign of significant under-fitting. On the other hand, there are 130 instances that Manthan solves, but neither CADET nor BFSS can solve. These instances have high dependencies between variables that Manthan can infer from the samples en route to predicting good candidate Skolem functions. Akshay et al. [4] suggest that C2Syn is an orthogonal approach to BFSS. Manthan solves 81 instances that neither C2Syn nor BFSS is able to solve, and these tools together solve 86 instances that Manthan fails to solve. Overall, Manthan solves **60** instances beyond the reach of any of the above state of the art tools.

6.2 Impact of the sampling scheme

To analyze the impact of the adaptive sampling and the quality of distributions generated by underlying samplers, we augmented Manthan with samples drawn from different samplers for adaptive and non-adaptive sampling. In particular, we employed QuickSampler [16], KUS [42], UniGen2 [15], and BiasGen[3]. The samplers KUS and UniGen2 could only produce samples for mere 14 and 49 benchmarks respectively within a timeout of 3600 seconds. Hence, we have omitted KUS and UniGen2 from further analysis. We also experimented with a naive enumeration of solution using off-the-shelf SAT solver, CryptoMiniSat [43]. It is worth noting that QuickSampler performs worse than BiasGen for uniformity testing using Barbarik [13]. In our implementation, we had to turn off the validation phase of QuickSampler to allow generation number of samples within a reasonable time. To statistically validate our intuition described in Section 4, we performed adaptive sampling using BiasGen. We use AdaBiasGen to refer to the adaptive sampling implementation.

Table 3 presents the performance of Manthan with different samplers listed in Column 1. The columns 2, 3, and 4 lists the number of instances that were solved during the execution of respective phases: Preprocess, LearnSkF, and Refine. Finally, column 5 lists the total number of instances solved. Two important findings emerge from Table 3: Firstly, as the quality of samplers improve, so does the performance of Manthan. In particular, we observe that with the improvement in

[3] BiasGen is developed by Mate Soos and Kuldeep S. Meel, and is pending publication.

the quality of samples leads to Manthan solving more instances in LearnSkF. Secondly, we see a significant increase in the number of instances that can be solved due to LearnSkF with samples from AdaBiasGen. It is worth remarking that one should view the adaptive scheme proposed in Section 4 to be a proof of concept and our results will encourage the development of more complex schemes.

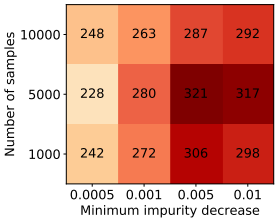


Fig. 3: Heatmap of # instances solved.(Best viewed in color)

Sampler	No. of instances solved			#Solved
	Preprocess	LearnSkF	Refine	
CryptoMiniSat	66	14	191	271
QuickSampler	66	28	181	275
BiasGen	66	51	228	345
AdaBiasGen	66	66	224	**356**

Table 3: Manthan with different samplers

6.3 Impact of LearnSkF

To analyze the impact of different design choices in LearnSkF, we analyzed the performance of Manthan for different samples (1000, 5000 and 10000) generated by GetSamples and for different choices of minimum impurity decrease (0.001, 0.005, 0.0005). Figure 3 shows a heatmap on the number of instances solved on each combination of the hyperparameters; the closer the color of a cell is to the red end of the spectrum, the better the performance of Manthan.

At the first look, Figure 3 presents a puzzling picture: It seems that increasing the number of samples does not improve the performance of Manthan. On a closer analysis, we found that the increase in the number of samples leads to an increase in the runtime of CandidateSkF but without significantly increasing the number of instances solved during LearnSkF. The runtime of CandidateSkF is dependent on the number of samples and $|Y|$. On the other hand, we see an interesting trend with respect to minimum impurity decrease where the performance first improves and then degrades. A plausible explanation for such a behavior is that with an increase in *minimum impurity decrease*, the generated decision trees tend to underfit while significantly low values of *minimum impurity decrease* lead to overfitting. We intend to study this in detail in the future.

Based on the above observations, we set the value of minimum impurity decrease to 0.005 and set the number of samples to (1) 10000 for $|Y| < 1200$, (2) 5000 for $1200 < |Y| \leq 4000$, and (3) 1000 for $|Y| > 4000$.

6.4 Division of time taken across different phases

To analyze the time taken by different phases of Manthan across different categories of the benchmarks, we normalize the time taken for each of the four

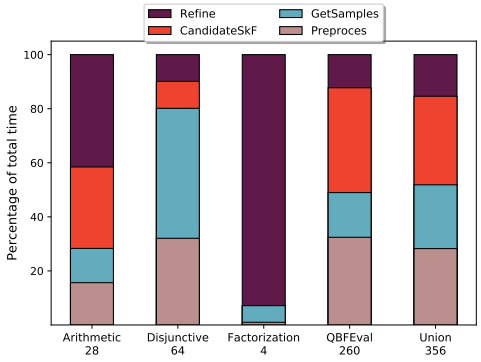


Fig. 4: Fraction of time spent in different phases in Manthan over different classes of benchmarks. (Best viewed in color)

core subroutines, Preprocess, GetSamples, CandidateSkF, and RefineSkF, for every benchmark that was solved by Manthan such that the sum of time taken for each benchmark is 1. We then compute the mean of the normalized times across different categories instances. Figure 4 shows the distribution of mean normalized times for different categories: Arithmetic, Disjunction, Factorization, QBFEval, and all the instances.

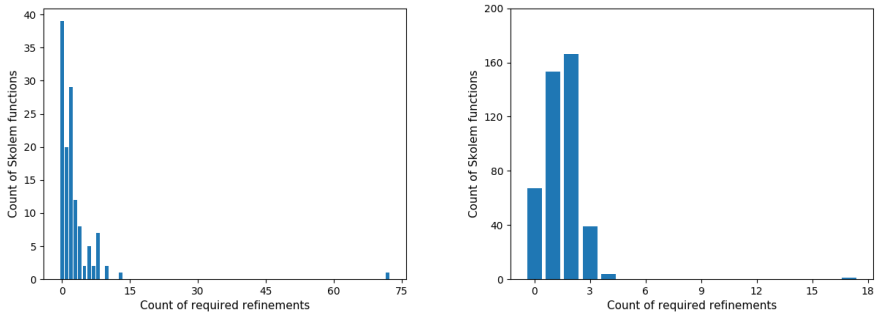
The diversity of our benchmark suite shows a nuanced picture and shows that the time taken by different phases strongly depends on the family of instances. For example, the disjunctive instances are particularly hard to sample and an improvement in the sampling techniques would lead to significant performance gains. On the other hand, a significant fraction of runtime is spent in the CandidateSkF subroutine indicating the potential gains due to improvement in decision tree generation routines. In all, Figure 4 identifies the categories of instances that would benefit from algorithmic and engineering improvements in Manthan's different subroutines.

6.5 Impact of using MaxSAT

In RefineSkF, Manthan invokes the MaxSATList subroutine, which calls MaxSAT solver to identify the potential erring Skolem functions. To observe the impact of using MaxSAT solver to identify the candidates to refine, we did an experiment with Manthan, without MaxSATList subroutine call. For all y_i, where $\sigma[y_i] \neq \sigma[y_i']$ were considered as candidates to refine. Manthan without MaxSATList subroutine call solved 204 instances that represents a significant drop in the number of solved instances by Manthan with MaxSATList subroutine.

6.6 Impact of self-substitution

To understand the impact of self-substitution, we profile the behavior of candidate Skolem functions with respect to number of refinements for two of our benchmarks; *pdtpmsmiim-all-bit* and *pdtpmsmiim*. In Figure 5, we use histograms



(a) Benchmark *pdtpmsmiim-all-bit:* plot for no. of Skolem functions vs required no. of refinements

(b) Benchmark *pdtpmsmiim:* plot for no. of Skolem functions vs required no. of refinements

Fig. 5: The plots to show the required number of refinements for the candidate Skolem functions.

with the number of candidate Skolem functions on y-axis and required number of refinements on x-axis. A bar of height a i.e $y = a$ at b i.e $x = b$ in Figure 5 represents that a candidate Skolem functions converged in b refinements. The histograms show that only a few Skolem functions require a large number of refinements: the tiny bar towards the right end in Figure 5a represents that for the benchmark *pdtpmsmiim-all-bit* only 1 candidate Skolem function required more than 60 refinements whereas all other candidate Skolem functions needed less than 15 refinements. Similarly, for the benchmark *pdtpmsmiim*, Figure 5b shows that only 1 candidate Skolem function was refined more than 15 times, whereas all other Skolem functions required less than 5 refinements. We found similar behaviors in many of our other benchmarks.

Based on the above trend and an examination of the decision trees corresponding to these instances, we hypothesize that some Skolem functions are hard to learn through data. For such functions, the candidate Skolem function generated from the data-driven phase in Manthan tends to be poor, and hence Manthan requires a long series of refinements for convergence. Since our refinement algorithm is designed for small, efficient corrections, we handle such hard to learn Skolem functions by synthesizing via self-substitution. Manthan detects such functions via a threshold on the number of refinements, which is empirically determined as 10, to identify hard to learn instances and sets them up for self-substitution.

In our experiments, we found 75 instances out of 356 solved instances required self-substitution, and for 51 of these 75 instances, only one variable undergoes self-substitution. Table 4 shows the impact of self-substitution for five of our benchmarks: Manthan has significant performance improvement with self-substitution in terms of the required number of refinements, which in turns affects the overall time. Note that Manthan can refine multiple candidates in a single RefineSkF call. For the first four benchmarks, all the other Skolem function

except the poor candidates were synthesized earlier than 10 refinement iteration, and at the 10^{th} refinement iteration the poor candidate functions hit our threshold for self-substitution. Taking the case of the last benchmark, all the other Skolem functions for it were synthesized earlier than 40 refinement cycles, and the last 16 iterations were only needed for 2 of the poor candidate functions to hit our threshold for self-substitution. Note that self-substitution can lead to an exponential blowup in the size of the formula, but it works quite well in our design as most Skolem functions are learnt quite well in the LearnSkF phase.

Table 4: Manthan : Impact of self substitution

Benchmarks $\exists Y F(X,Y)$	$\lvert X\rvert$	$\lvert Y\rvert$	No. of Refinements		Time(s)	
			Self-Substitution		Self-Substitution	
			Without	With	Without	With
kenflashpo2-all-bit	71	32	319	10	35.88	19.22
eijkbs1512	316	29	264	10	42.88	32.35
pdtpmsmiim-all-bit	429	30	313	10	72.75	36.08
pdtpmssfeistel	1510	68	741	10	184.11	115.07
pdtpmsmiim	418	337	127	56	1049.29	711.48

7 Conclusion

Boolean functional synthesis is a fundamental problem in Computer Science with a wide variety of applications. In this work, we propose a novel data-driven approach to synthesis that employs constrained sampling techniques for generation of data, machine learning for candidate Skolem functions, and automated reasoning to verify and refine to generate Skolem functions. Our approach achieves significant performance improvements. As pointed out in Section 5 and 6, our work opens up several interesting directions for future work at the intersection of machine learning, constrained sampling, and automated reasoning.

Acknowledgment We are grateful to the anonymous reviewers and Dror Fried for constructive comments that significantly improved the final version of the paper. We are grateful to Mate Soos for tweaking BiasGen to support Manthan. We are indebted to S. Akshay, Supratik Chakraborty, and Shetal Shah for their patient responses to our tens of queries regarding prior work.

This work was supported in part by National Research Foundation Singapore under its NRF Fellowship Programme [NRF-NRFFAI1-2019-0004] and AI Singapore Programme [AISG-RP-2018-005], and NUS ODPRT Grant [R-252-000-685-13]. The computational work for this article was performed on resources of the National Supercomputing Centre, Singapore: https://www.nscc.sg [1].

References

1. ASTAR, NTU, NUS, SUTD: National Supercomputing Centre (NSCC) Singapore (2018), https://www.nscc.sg/about-nscc/overview/
2. QBF solver evaluation portal 2017, http://www.qbflib.org/qbfeval17.php
3. QBF solver evaluation portal 2018, http://www.qbflib.org/qbfeval18.php
4. Akshay, S., Arora, J., Chakraborty, S., Krishna, S., Raghunathan, D., Shah, S.: Knowledge compilation for boolean functional synthesis. In: Proc. of FMCAD (2019)
5. Akshay, S., Chakraborty, S., Goel, S., Kulal, S., Shah, S.: Whats hard about boolean functional synthesis? In: Proc. of CAV (2018)
6. Akshay, S., Chakraborty, S., John, A.K., Shah, S.: Towards parallel boolean functional synthesis. In: Proc. of TACAS (2017)
7. Alur, R., Bodik, R., Juniwal, G., Martin, M.M., Raghothaman, M., Seshia, S.A., Singh, R., Solar-Lezama, A., Torlak, E., Udupa, A.: Syntax-guided synthesis. In: Proc. of FMCAD (2013)
8. Balabanov, V., Jiang, J.H.R.: Resolution proofs and skolem functions in QBF evaluation and applications. In: Proc. of CAV (2011)
9. Balabanov, V., Jiang, J.H.R.: Unified QBF certification and its applications. In: Proc. of FMCAD (2012)
10. Bavishi, R., Pandey, A., Roy, S.: To be precise: regression aware debugging. In: Proc. of OOPSLA (2016)
11. Biere, A.: PicoSAT essentials. Proc. of JSAT (2008)
12. Boole, G.: The mathematical analysis of logic. Philosophical Library (1847)
13. Chakraborty, S., Meel, K.S.: On testing of uniform samplers. In: Proc. of AAAI (2019)
14. Chakraborty, S., Fried, D., Tabajara, L.M., Vardi, M.Y.: Functional synthesis via input-output separation. In: Proc. of FMCAD (2018)
15. Chakraborty, S., Meel, K.S., Vardi, M.Y.: Balancing scalability and uniformity in SAT witness generator. In: Proc. of DAC (2014)
16. Dutra, R., Laeufer, K., Bachrach, J., Sen, K.: Efficient sampling of SAT solutions for testing. In: Proc. of ICSE (2018)
17. Ezudheen, P., Neider, D., D'Souza, D., Garg, P., Madhusudan, P.: Horn-ICE learning for synthesizing invariants and contracts. In: Proc. of OOPSLA (2018)
18. Fedyukovich, G., Gupta, A.: Functional synthesis with examples. In: Proc. of CP (2019)
19. Fried, D., Tabajara, L.M., Vardi, M.Y.: BDD-based boolean functional synthesis. In: Proc. of CAV (2016)
20. Garg, P., Löding, C., Madhusudan, P., Neider, D.: ICE: A robust framework for learning invariants. In: Proc. of CAV (2014)
21. Garg, P., Neider, D., Madhusudan, P., Roth, D.: Learning invariants using decision trees and implication counterexamples. In: Proc. of POPL (2016)
22. Golia, P., Roy, S., Meel, K.S.: Manthan: A data driven approach for boolean function synthesis (2020), https://arxiv.org/abs/2005.06922
23. Grumberg, O., Lerda, F., Strichman, O., Theobald, M.: Proof-guided underapproximation-widening for multi-process systems. In: Proc. of POPL (2005)
24. Gupta, R., Sharma, S., Roy, S., Meel, K.S.: WAPS: Weighted and projected sampling. In: Proc. of TACAS (2019)
25. Heule, M.J., Seidl, M., Biere, A.: Efficient extraction of skolem functions from QRAT proofs. In: Proc. of FMCAD (2014)

26. Janota, M.: Towards generalization in QBF solving via machine learning. In: Proc. of AAAI (2018)
27. Jo, S., Matsumoto, T., Fujita, M.: SAT-based automatic rectification and debugging of combinational circuits with lut insertions. Proc. of IPSJ T-SLDM (2014)
28. John, A.K., Shah, S., Chakraborty, S., Trivedi, A., Akshay, S.: Skolem functions for factored formulas. In: Proc. of FMCAD (2015)
29. Jose, M., Majumdar, R.: Cause clue clauses: error localization using maximum satisfiability. In: Proc. of PLDI (2011)
30. Jussila, T., Biere, A., Sinz, C., Kröning, D., Wintersteiger, C.M.: A first step towards a unified proof checker for QBF. In: Proc. of SAT (2007)
31. Logic, B., Group, V.: ABC: A system for sequential synthesis and verification, http://www.eecs.berkeley.edu/~alanmi/abc/
32. Lonsing, F., Egly, U.: Depqbf 6.0: A search-based QBF solver beyond traditional QCDCL. In: Proc. of CADE (2017)
33. Löwenheim, L.: Über die auflösung von gleichungen im logischen gebietekalkul. Mathematische Annalen (1910)
34. Martins, R., Manquinho, V., Lynce, I.: Open-WBO: A modular MaxSAT solver. In: Proc. of SAT (2014)
35. Massacci, F., Marraro, L.: Logical cryptanalysis as a SAT problem. Journal of Automated Reasoning (2000)
36. Niemetz, A., Preiner, M., Lonsing, F., Seidl, M., Biere, A.: Resolution-based certificate extraction for QBF. In: Proc. of SAT (2012)
37. Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., Duchesnay, E.: Scikit-learn: Machine Learning in Python. Proc. of Machine Learning Research (2011)
38. Quinlan, J.R.: Induction of decision trees. Proc. of Machine learning (1986)
39. Rabe, M.N.: Incremental determinization for quantifier elimination and functional synthesis. In: Proc. of CAV (2019)
40. Rabe, M.N., Tentrup, L.: CAQE: A certifying QBF solver. In: Proc. of FMCAD (2015)
41. Rabe, M.N., Tentrup, L., Rasmussen, C., Seshia, S.A.: Understanding and extending incremental determinization for 2QBF. In: Proc. of CAV (2018)
42. Sharma, S., Gupta, R., Roy, S., Meel, K.S.: Knowledge compilation meets uniform sampling. In: Proc. of LPAR (2018)
43. Soos, M.: msoos/cryptominisat (2019), https://github.com/msoos/cryptominisat
44. Srivastava, S., Gulwani, S., Foster, J.S.: Template-based program verification and program synthesis. STTT (2013)
45. Tabajara, L.M., Vardi, M.Y.: Factored boolean functional synthesis. In: Proc. of FMCAD (2017)
46. Verma, S., Roy, S.: Synergistic debug-repair of heap manipulations. In: Proc. of ESEC/FSE (2017)