Engineering an Efficient Boolean Functional Synthesis Engine

Priyanka Golia
Indian Institute of Technology Kanpur
National University of Singapore

Friedrich Slivovsky
TU Wien

Subhajit Roy
Indian Institute of Technology Kanpur

Kuldeep S. Meel
National University of Singapore

***Abstract*—Given a Boolean specification between a set of inputs and outputs, the problem of Boolean functional synthesis is to synthesise each output as a function of inputs such that the specification is met. Although the past few years have witnessed intense algorithmic development, accomplishing scalability remains the holy grail. The state-of-the-art approach combines machine learning and automated reasoning to synthesise Boolean functions efficiently. In this paper, we propose four algorithmic improvements for a data-driven framework for functional synthesis: using a dependency-driven multi-classifier to learn candidate function, extracting uniquely defined functions by interpolation, variables retention, and using lexicographic MaxSAT to repair candidates.**

We implement these improvements in the state-of-the-art framework, called Manthan. The proposed framework is called Manthan2. Manthan2 shows significantly improved runtime performance compared to Manthan. In an extensive experimental evaluation on 609 benchmarks, Manthan2 is able to synthesise a Boolean function vector for 509 instances compared to 356 instances solved by Manthan – an increment of 153 instances over the state-of-the-art. To put this into perspective, Manthan improved on the prior state-of-the-art by only 76 instances.

I. Introduction

Given two sets $X = \{x_1, \ldots, x_n\}$ and $Y = \{y_1, \ldots, y_m\}$ of variables and a Boolean formula $F(X, Y)$ over $X \cup Y$, the problem of Boolean functional synthesis is to compute a vector $\Psi = \langle \psi_1, \ldots, \psi_m \rangle$ of Boolean functions ψ_i (often called *Skolem functions*) such that $\exists Y F(X, Y) \equiv F(X, \Psi(X))$. Informally, given a specification between inputs and outputs, the task is to synthesise a function Ψ that maps each assignment of the inputs to an assignment of the outputs so that the combined assignment meets the specification (whenever such an assignment exists). With origins tracing to Boole's seminal work [12], functional synthesis is a fundamental problem in computer science that has a wide variety of applications in areas such as circuit synthesis [33], program synthesis [52], automated program repair [31], cryptography [40], logic minimization [13], [14]. For example, the relation F can specify the allowed behavior of a circuit of interest and the function Ψ corresponds to the implementation of the desired circuit. As pointed out by Jiang, Lin, and Hung [30], relations can succinctly capture the conventional notion of *don't cares*. Furthermore, extracting functions from Boolean relations also has applications in 2-level logic minimization under the Sum-of-Products (SOP) representation [20], [28], [35].

Over the past two decades, functional synthesis has seen a surge of interest, leading to the development of new approaches that can be broadly classified into three categories: 1) incremental determinization iteratively identifies variables with unique Skolem functions and takes "decisions" on any remaining variables by adding temporary clauses that make them deterministic [42], [43], [45]. 2) Skolem functions can be obtained by eliminating quantifiers using functional composition, and Craig interpolation can be applied to reduce the size of composite functions [29], [30]. Although this typically does not scale to large specifications, it was shown to work well using ROBDDs in combination with carefully chosen variable orderings [17], [53]. 3) CEGAR-style approaches start from an initial set of approximate Skolem functions, followed by a phase of counter-example guided refinement to patch these candidate functions [5], [6], [32]. With the right choice of initial functions, the CEGAR phase can often be skipped entirely, a phenomenon that can be analyzed in terms of knowledge compilation [4], [5].

Recently, we proposed a new data-driven approach Manthan [21]. Manthan relies on constrained sampling [51] to generate satisfying assignments of the formula F, which are fed to a decision-tree learning technique such that the learned classifiers represent potential Skolem functions, called candidates. The candidates are repeatedly tested for correctness and repaired in a subsequent CEGAR loop, with a MaxSAT solver minimizing the number of repairs required for each counterexample. While Manthan achieved significant improvement of the state-of-the-art, a large number of problems remain beyond its reach (and other synthesis engines).

The primary contribution of this work is to address scalability barriers faced by Manthan. To this end, we propose the following four crucial algorithmic innovations:

1) **Interpolation-based Unique Function Extraction**: We identify a subset of variables with *unique Skolem function* and extract these functions by interpolation, thereby reducing the number of functions that need to be learned.
2) **Clustering-based Multi-Classification**: We propose a clustering-based approach that can take advantage of multi-classification to learn candidate functions for sets of variables at a time.
3) **Learning and Repair over Determined Features**: Whenever it is determined that a candidate function for a variable is indeed a Skolem function, we do not substitute for and eliminate this variable, and instead retain it as a possible feature during learning and repair. Our strategy stands in stark contrast to the conventional wisdom that advocates variable elimination.
4) **Lexicographic MaxSAT-based Dependency-Aware Repair**: We design a lexicographic MaxSAT-based strategy for identifying repair candidates so as to take into account dependencies among candidate functions.

To measure the impact of these proposed algorithmic innovations, we implemented them in a system named Manthan2 and performed an extensive evaluation on a benchmark suite used in prior studies [4], [5], [21]. In terms of solved instances, the results are decisive. Out of 609 instances, Manthan and CADET are able to solve 356 and 280 instances, in line with experimental results reported in prior work that saw a 76 instance lead of Manthan over the then state-of-the-art CADET [21]. Manthan2 solves 509 instances and thereby achieves a dramatic improvement of 153 instances over Manthan, more than doubling the already substantial increase in the number of solved instances achieved by Manthan over CADET.

The rest of the paper is organized as follows: In Section II, we first introduce notation and then provide some background on Manthan.In Section III, we present an overview of the invocations implemented in Manthan2, before giving a detailed algorithmic description in

Manthan2 is available at https://github.com/meelgroup/manthan

Section IV. We then describe the experimental methodology and discuss results with respect to each of the technical contributions of Manthan2 in Section V. We cover related work in Section VI.

II. Background

We use lower case letters to denote propositional variables and capital letters to denote sets of variables. Given a set $\{v_1, \ldots, v_n\}$ of variables and $1 \leq i \leq j \leq n$, we write V_i^j for the subset $\{v_i, v_{i+1}, \ldots, v_j\}$. We use standard notation for logical connectives such as $\wedge, \vee$ and $\neg$. A *literal* is a variable or a negated variable. A formula φ is in Conjunctive Normal Form (CNF) if it is a conjunction of *clauses*, where each *clause* is a disjunction of literals. We write $Vars(\varphi)$ to denote the set of variables used in φ. A *satisfying assignment* of a formula φ is a mapping $\sigma : Vars(\varphi) \rightarrow \{0, 1\}$ such that φ evaluates to True under σ. We write $\sigma \models \varphi$ to denote that σ is a satisfying assignment of φ. Given a subset V of variables, we write $\sigma[V]$ to denote the restriction of σ to V. An *unsatisfiable core* of a formula in CNF is a subset of clauses for which there is no satisfying assignment. We use UnsatCore to denote an unsatisfiable core when the formula is understood from the context.

For a given CNF formula in which some clauses are declared as *hard constraints* and the rest are declared as *soft constraints*, the problem of (partial) MaxSAT is to find an assignment of the given formula that satisfies all hard constraints and maximizes the number of satisfied soft constraints. Furthermore, *lexicographic* partial MaxSAT, or LexMaxSAT for short, is a special case of partial MaxSAT where there is a preference in the order in which to satisfy the soft constraints.

A. Functional Synthesis

We assume a relational specification $\exists Y F(X, Y)$ such that $X = \{x_1, \ldots, x_n\}$ and $Y = \{y_1, \ldots, y_m\}$. We write F and $F(X, Y)$ interchangeably, and use $F(X, Y)|_{y_i=b}$ to denote the result of substituting $b \in \{0, 1\}$ for y_i in $F(X, Y)$.

*a) **Problem Statement**:* Given a specification $\exists Y F(X, Y)$ with inputs X and outputs Y, the task of *Boolean functional synthesis* is to find a function vector $\Psi = \langle \psi_1, \ldots, \psi_m \rangle$ such that $\exists F(X, Y) \equiv F(X, \Psi(X))$. We refer to Ψ as a *Skolem function vector* and to the function ψ_i as a *Skolem function* for y_i.

We solve a slightly relaxed version of this problem by synthesising a Skolem function vector $\langle \psi_1, \ldots, \psi_m \rangle$ such that $y_i = \psi_i(X, y_1, \ldots, y_{i-1})$ for a given order $y_1, \ldots, y_m$; this is ultimately equivalent to synthesising $\Psi(X)$, since each $\psi_i(X, y_1, \ldots, y_{i-1})$ can be transformed into a function depending only on X by substituting the functions for $y_1, \ldots, y_{i-1}$. We write $\prec_d$ to denote the (smallest) partial order on the output variables Y such that $y_i \prec_d y_j$ if y_j appears in ψ_i, and say that y_i *depends on* y_j whenever $y_i \prec_d y_j$.

B. Definability

Definition 1 ([34]): Let $F(W)$ be a formula, $w \in W$, $S \subseteq W \setminus w$. $F(W)$ *defines* w in terms of S if and only if there exists a formula $H(S)$ such that $F(W) \models w \leftrightarrow H(S)$. In such a case, $H(S)$ is called a *definition* of w on S in $F(W)$.

To this end, given $F(W)$ defined on $W = \{w_1, w_2, \ldots w_n\}$. We create another set of *fresh* variables $Z = \{z_1, z_2, \ldots z_n\}$. Let $F(W \mapsto Z)$ represent the formula where every $w_i \in W$ in F is replaced by $z_i \in Z$.

Lemma 1 (Padoa's Theorem):

$$\text{Let, } I(W, Z, S, i) = F(W) \wedge F(W \mapsto Z) \wedge \left(\bigwedge_{w_j \in S; j \neq i} (w_j \leftrightarrow z_j) \right) \wedge w_i \wedge \neg z_i$$

F defines $w_i \in W$ in terms of S if and only if $I(W, Z, S, i)$ is UNSAT.

C. Manthan: Background

We now give a brief overview of the state-of-the-art Boolean functional synthesis tool Manthan [21]. Given a specification, Manthan computes Skolem functions in several phases described below.

*a) **Preprocessing:*** A variable y_i is *positive unate* (resp. *negative unate*) in $F(X, Y)$, if $F(X, Y)|_{y_i=0} \wedge \neg F(X, Y)|_{y_i=1}$ (resp. $F(X, Y)|_{y_i=1} \wedge \neg F(X, Y)|_{y_i=0}$) is UNSAT [5]. The Skolem function for a positive unate (resp. negative unate) variable y_i is the constant function $\psi_i = 1$ (resp. $\psi_i = 0$). Manthan finds unates as a preprocessing step.

*b) **Learning Candidates:*** Manthan adopts an *adaptive* weighted sampling strategy to sample satisfying assignments of $F(X, Y)$, which are used to learn decision tree classifiers. More specifically, Manthan samples uniformly over the input variables X while biasing the output variables Y towards a particular value. With the data generated, Manthan learns approximate candidate functions using a dependency driven binary classifier. To learn a candidate function ψ_i corresponding to y_i, Manthan considers the value of y_i in a satisfying assignment as a label, and values of $X \cup \hat{Y}$ as a feature set to construct a decision tree dt, where $\hat{Y}$ is the set of Y variables, such that for y_j of $\hat{Y}$, $y_j \not\prec_d y_i$. From the learned decision tree dt, Manthan obtains the candidate function as the disjunction of all the paths with leaf node label 1. For every y_k occurring as decision node in dt, Manthan updates the dependencies as $y_i \prec_d y_k$. Finally, Manthan extends the partial order $\prec_d$ to get a *TotalOrder* of Y variables.

*c) **Verification:*** Manthan checks if the learned candidates are Skolem functions or not by checking satisfiability of the *error formula* $E(X, Y, Y')$ defined as

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

where $Y' = \{y'_1, \ldots, y'_m\}$ is a set of fresh variables. It is readily verified that Ψ is a Skolem function vector if, and only if, $E(X, Y, Y')$ is UNSAT [32]. If $E(X, Y, Y')$ is SAT and $\sigma \models E(X, Y, Y')$, then Manthan has a counterexample σ to fix.

*d) **Repairing Candidates:*** Manthan finds candidate functions to repair by making a MaxSAT call with hard constraints $F(X, Y) \wedge (X \leftrightarrow \sigma[X])$ and soft constraints $(Y' \leftrightarrow \sigma[Y'])$. The output variables associated with the soft constraints that are *not* satisfied form a smallest subset of output variables whose candidate functions need to change to satisfy the specification. Now, to repair a candidate function ψ_i corresponding to output variable y_i, Manthan constructs another formula $G_i(X, Y)$ as

$$G_i(X, Y) := (y_i \leftrightarrow \sigma[y'_i]) \wedge F(X, Y) \wedge (X \leftrightarrow \sigma[X]) \wedge (\hat{Y} \leftrightarrow \sigma[\hat{Y}]), \quad (2)$$

where $\hat{Y} \subset Y$ is the set $\hat{Y} = \{\textit{TotalOrder}[index(y_i) + 1], \cdots, \textit{TotalOrder}[|Y|]\}$.

If $G_i(X, Y)$ turns out to be UNSAT, then Manthan constructs a repair formula β as the conjunction of all unit clauses of an UnsatCore of $G_i(X, Y)$. Depending on the current valuation of the candidate function ψ, Manthan strengthens or weaken the candidate by the repair formula β. Otherwise, if $G_i(X, Y)$ is SAT, Manthan looks for other candidate functions to repair instead.

During the repair phase, Manthan uses self-substitution [29] as a fallback: whenever more than 10 iterations are needed for repairing a particular candidate, Manthan directly synthesises a Skolem function for that variable via self-substitution.

III. Overview

In this section, we provide an overview of our primary contributions in Manthan2, building on the Manthan [21] infrastructure.

A. Interpolation-based Unique Function Extraction

In order to reduce Manthan's reliance on data-driven learning, we seek to identify a subset $Z \subseteq Y$ and the corresponding Skolem function vector Φ such that Φ can be extended to a valid Skolem function vector Ψ. In the following, we call such a Z a *determined set*. Observe that unate variables form a determined set Z. To grow Z further, we rely on the notion of definability, and iteratively identify the variables $y_i \in Y$ such that y_i is definable in terms of rest of the variables such that its definition ψ_i respects the dependency constraints imposed by the definitions of variables in Z. To extract the corresponding definitions, we rely on the Padoa's theorem (Lemma 1) to check whether y_i is definable in terms of rest of the variables and then employ interpolation-based extraction of the corresponding definition [49].

The usage of unique function extraction significantly reduces the number of variables for which Manthan2 needs to learn and repair the candidates since unique functions do not need to undergo refinement. While our primary motivation for unique function extraction was to reduce the over-reliance on learning, it is worth emphasizing that interpolation-based extraction is also able to compute complicated functions with large size; these functions would require a prohibitive number of samples and as such lie beyond the scope of a practical learning-based technique.

We close by highlighting the importance of allowing y_i to depend, subject to dependency constraints, on other Y variables. Consider, $X = \{x_1\}$ and $Y = \{y_1, y_2\}$, and let $F(X, Y) := (y_1 \vee y_2) \wedge (\neg y_1 \vee \neg y_2)$. Neither y_1 nor y_2 is defined by x_1. But y_2 is definable in terms of $\{y_1\}$ (and therefore, also $\{x_1, y_1\}$) with its corresponding Skolem function $\psi_2(x_1, y_1) := \neg y_1$.

Impact: For over 40% of our benchmarks, Manthan2 was able to extract Skolem functions for at least 95% of total variables via unique function extraction.

B. Learning and Repair over Determined Features

As mentioned in the previous section, we focus on constructing a determined set Z consisting of unates and variables with unique functions. All the variables in Z can be eliminated by substituting them with their corresponding definitions (in case of unates, the definition is a constant: True or False). Variable elimination has a long history as an effective preprocessing strategy [4], [5], [11], [21], and, following this tradition, Manthan performs variable elimination wherever possible. In particular, it eliminates unates as well as variables for which definitions can be obtained via syntactic gate extraction techniques.

While substituting for variables in Z does not affect the *existence* of Skolem functions for variables $y_i \in Y \setminus Z$, the *size* of these functions can increase substantially when they are not allowed to depend on variables in Z. We also observe that variables in Z can considered as *determined features* and the Skolem functions for some $y_i \in Y \setminus Z$ can be efficiently represented in Z. For example, consider the following scenario: let $X = \{x_1, x_2\}$, $Y = \{y_1, y_2\}$ and $F(X, Y) = (y_1 \vee y_2) \wedge (\neg y_1 \vee \neg y_2) \wedge (y_1 \leftrightarrow (x_1 \oplus x_2))$. Observe that the Skolem function for y_2 in terms of X in the transformed formula $F(x_1, x_2, y_2)$ will have to be learned as $\neg(x_1 \oplus x_2)$. However, when allowing learning over y_1, then the desired Skolem function for y_2 can simply be learned as $\neg y_1$.[1]

Further, every iteration of our repair phase adds clauses over the literals in the formula, and therefore allowing a repair clause to contain a variable $y_i \in Z$ with definition ψ_i increases the expressiveness of the clauses during the repair phase, akin to bounded variable addition.

We conclude that, contrary to conventional wisdom, variables in the determined set Z should not be eliminated and instead should be retained as features for the learning and repair phases of Manthan2.

Impact: The retention of variables in the determined set allows Manthan2 to solve 25 more benchmarks.

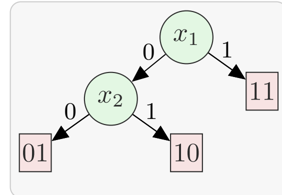


Fig. 1: Multi-Classification: learned decision tree with labels $\{y_1, y_2\}$ and features $\{x_1, x_2\}$

C. Clustering-based Multi-Classification

For some of the benchmarks, Manthan spends $\sim 74\%$ of its time in learning the candidate functions. To reduce this learning time, Manthan2 uses the following strategy:

1) Partition the set of Y variables into disjoint subsets,
2) Use a multi-classifier (instead of a binary-classifier) to learn candidate Skolem functions for each partition.

For example, let $X = \{x_1, x_2\}$ and $Y = \{y_1, y_2\}$ in $\exists Y F(X, Y)$. Figure 1 shows the learned decision tree with labels $\{y_1, y_2\}$, and features $\{x_1, x_2\}$. The expected number of classes to learn two Y variables is $2^2 = 4$, but as shown in Figure 1, the decision tree classifies the labels into 3 classes $\langle 01, 10, 11\rangle$. The candidate function ψ_1 corresponding to y_1 is the disjunction of paths from root to leaf node with label of y_1 being 1, i.e, the classes 10 and 11. Hence, the candidate function $\psi_1 := (\neg x_1 \wedge x_2) \vee (x_1)$. Similarly, the candidate function ψ_2 for y_2 is $\psi_2 := (\neg x_1 \wedge \neg x_2) \vee (x_1)$.

The candidate Skolem function for a variable y_i of a chosen subset is obtained as the disjunction of all the paths from the root to leaf node with a label of y_i being 1. We further update the partial dependency as $y_i \prec_d y_j$, for all y_j variables occurring in ψ_i. Now, let us consider the case with two different subsets $\{y_1, y_2\}$ and $\{y_3, y_4\}$, and also assume that $y_1 \prec_d y_3$, then the feature set to learn $\{y_3, y_4\}$ would be $\{X, y_2\}$. The feature set to learn a chosen subset would include a variable y_j, only if $y_j \not\prec_d y_i$ for every variable y_i of the subset.

An important question that remains to be answered is *how should the variable partitioning be driven*? The intuition behind our approach lies in the fact that low cohesion among variables in a partition would impose fewer constraints, leading to larger trees and multiplying the number of classes. Therefore, in some sense, we would like to learn *related* variables together. Manthan2 uses the distance in the *primal graph* [47] to cluster Y variables into disjoint subsets, such that variables in a subset are closely related.

[1] There is an analogy with the role of *latent features* in machine learning, which allow for the compact representation of a model but must first be computed from observable features: elimination of variables with unique Skolem functions turns observable features into latent features that must be recovered by the learning algorithm.

Impact: We observe a decrease of 252 seconds in the PAR-2 score by using a multi-classifier to learn a subset of variables together over learning one candidate at a time.

D. Lexicographic MaxSAT-based Dependency-Aware Repair

Let us start by demonstrating a troublesome scenario for Manthan on the same running example as above: $X = \{x_1, x_2\}$ and $Y = \{y_1, y_2\}$ and let $F(X,Y) = (y_1 \vee y_2) \wedge (\neg y_1 \vee \neg y_2)$ in $\exists Y F(X,Y)$, with the candidates $\psi_1 = 1$ and $\psi_2 = 1$, and *TotalOrder* = $\{y_1, y_2\}$. As the candidates are not yet Skolem functions, Manthan starts off by identifying a candidate for repair by invoking MaxSAT with hard constraints $F(X,Y) \wedge (X \leftrightarrow \sigma[X])$ and soft constraints $(y_1 \leftrightarrow 1) \wedge (y_2 \leftrightarrow 1)$, where σ is a satisfying assignment of the error formula (1). As either y_1 or y_2 can be flipped to fix the counterexample σ, let us assume MaxSAT does not satisfy the soft constraint $(y_2 \leftrightarrow 1)$, thereby selecting ψ_2 for repair.

In order to repair ψ_2, Manthan constructs the formula G_2 (2) as $G_2 = F(X,Y) \wedge (X \leftrightarrow \sigma[X]) \wedge (y_2 \leftrightarrow 1)$. As G_2 is not allowed to constrain over y_1, it turns out as SAT, hence adding ψ_1 as a candidate to repair. Therefore, to fix the counterexample σ, Manthan fails to repair the candidate, and requires an additional repair iteration. This scenario could have been averted and the counterexample σ can be fixed in the same repair iteration if ψ_1 was selected before ψ_2

Manthan2 uses LexMaxSAT [27] to satisfy the soft constraints in accordance to the *TotalOrder*. For the aforementioned problem, if the soft constraint $y_1 \leftrightarrow 1$ takes preference over $y_2 \leftrightarrow 1$, Manthan2 would pick candidate corresponding to y_1 as a repair candidate. Therefore, the use of LexMaxSAT in finding repair candidates reduces the required number of iterations to fix a counterexample.

However, LexMaxSAT can be expensive [7], [38]. To avoid frequent LexMaxSAT calls, Manthan2 first computes a list of candidates to repair using unweighted MaxSAT. This list can grow whenever a formula G_i turns out to be SAT. Once its size exceeds a certain threshold, Manthan2 recomputes another set of repair candidates using LexMaxSAT. In particular, LexMaxSAT is used only if Manthan2 has to fix many candidates in a single repair iteration due to an ordering constraint.

Impact: We observe a decrease of more than 100 seconds in the PAR-2 score by using LexMaxSAT.

IV. Algorithm

In this section, we present a detailed algorithmic description of Manthan2. Manthan2 takes a formula $F(X,Y)$, and returns a Skolem function vector. Manthan2 considers fixed values for k and s, where k is the maximum edge distance that is used to cluster Y variables together, and s is the maximum number of Y variables that can be learned together.

Manthan2 is presented in Algorithm 1, it starts off by extracting Skolem functions for unates and uniquely defined variables of the formula $F(X,Y)$ at line 3. The set U represents all the Y variables that are either unate or have unique Skolem functions. At line 4, Manthan2 generates the required number of samples. Next, at line 5, Manthan2 calls subroutine ClusterY to cluster the Y variables that are not in U. ClusterY returns a list, subsetY, that represents different subsets of Y variables for which the candidates would be learned together. To learn the candidate functions for each subsets, Manthan2 calls subroutine CandidateSkF at line 7. CandidateSkF also updates the dependencies among Y variables as per the learned candidate functions. Manthan2 now finds a total order *TotalOrder* of Y variables in accordance with dependencies among the Y variables at line 8. Manthan2 then checks the satisfiability of the error formula $E(X,Y,Y')$, and if $E(X,Y,Y')$ is SAT, it calls subroutine FindRepairCandidates to find the list of candidates to repair at line 13. Then at line 15, it calls subroutine RepairSkF to repair the candidates. This process is continued until the error formula $E(X,Y,Y')$ is UNSAT, and then, Manthan2 returns a Skolem function vector. Note that if $U = Y$, that is, if all Y variables are either unate or uniquely defined, then Manthan2 terminates after UniDef.

Manthan2 uses subroutines GetSamples, FindOrder and RepairSkF as described in [21].[2] And like Manthan, Manthan2 uses self-substitution [29] as a fallback (see Section II). We will now discuss the newly introduced subroutines.

1) UniDef: Algorithm 2 presents the subroutine UniDef. It assumes access to the following two subroutines:

1) FindUnates, which takes a formula $F(X,Y)$ as input and returns a list of unates and their corresponding Skolem functions.
2) FindUniqueDef, which takes a formula $F(X,Y)$, a variable y_i, and a defining set $X, y_1, \ldots, y_{i-1}$ as input, and determines whether the given variable y_i is defined with respect to the defining set or not. If the variable y_i is defined, FindUniqueDef returns true, along with the extracted definition ψ_i. Otherwise, it returns false (and an empty definition).

UniDef first calls FindUnates to find the unates and their corresponding Skolem functions at line 1. Then, it calls subroutine FindUniqueDef with defining set $\{X, y_1, \ldots, y_{i-1}\}$ for each existentially quantified variable y_i which is not unate at line 5. If FindUniqueDef returns true, UniDef adds y_i to the set ***univar*** at line 7. UniDef adds variables occurring in ψ_i to the list dependson$[y_i]$ at line 10.

2) ClusterY: Algorithm 3 presents the subroutine ClusterY, it takes formula the $F(X,Y)$, k : an edge distance parameter, s : maximum allowed size of a cluster of Y variables, and U : list of unate and uniquely defined Y variables, and it returns a list of all subsets of Y that would be learned together. ClusterY assumes access to a subroutine kHopNeighbor, which takes a graph, a variable y, and an integer k as input, and returns all variables within distance k of y in the graph.

[2]Note that the subroutines FindRepairCandidates, and RepairSkF are referred to as MaxSATList and RefineSkF in [21].

Algorithm 1 Manthan2(F(X,Y))

1: $\Psi \leftarrow \{\psi_1 = \emptyset, \ldots, \psi_{|Y|} = \emptyset\}$
2: dependson $\leftarrow \{\}$
3: U, Ψ, dependson $\leftarrow$ UniDef(F(X,Y),Ψ,dependson)
4: $\Sigma \leftarrow$ GetSamples(F(X,Y))
5: subsetY $\leftarrow$ ClusterY(F(X,Y),k,s,U)
6: **for** each chunk $\in$ subsetY **do**
7:     Ψ, dependson $\leftarrow$ CandidateSkF(Σ,F(X,Y), Ψ, chunk, dependson)
8: *TotalOrder* $\leftarrow$ FindOrder(dependson)
9: **repeat**
10:     $E(X,Y,Y') \leftarrow F(X,Y) \wedge \neg F(X,Y') \wedge (Y' \leftrightarrow \Psi)$
11:     ret, $\sigma \leftarrow$ CheckSat(E(X,Y,Y'))
12:     **if** ret = SAT **then**
13:         ind $\leftarrow$ FindRepairCandidates(F(X,Y),σ,*TotalOrder*)
14:         **for** $y_k \in$ ind **do**
15:             $\Psi \leftarrow$ RepairSkF(F(X,Y),σ,Ψ,*TotalOrder*)
16: **until** ret = UNSAT
17: **return** Ψ

Algorithm 2 UniDef(F(X,Y),Ψ,dependson)

Ψ, unates $\leftarrow$ FindUnates(F(X,Y))
univar $\leftarrow \emptyset$
for $y_i \in Y\setminus$ unates **do**
  definingvar $\leftarrow X \cup \{y_1, \ldots, y_{i-1}\}$
  ret, def $\leftarrow$ FindUniqueDef(F(X,Y),y_i,definingvar)
  if ret = true **then**
    univar $\leftarrow$ univar $\cup\ y_i$
    $\psi_i \leftarrow$ def
    for $y_j \in \psi_i$ **do**
      dependson[y_i] $\leftarrow$ dependson[y_i] $\cup\ y_j$
return unates $\cup$ univar, Ψ, dependson

Algorithm 3 ClusterY(F(X,Y),k,s,U)

1: graph = $\emptyset$
2: **for** each *clause* of $F(X,Y)$ **do**
3:   **if** $\langle y_i, y_j\rangle$ pair in *clause* **then**
4:     **if** $y_i \notin U$ and $y_j \notin U$ **then**
5:       AddEdge(graph,y_i,y_j)
6: subsetY = $\emptyset$
7: **for** $y_i \in Y$ **do**
8:   **while** $k \geq 0$ **do**
9:     chunk $\leftarrow$ kHopNeighbor(graph,y_i,k)
10:     **if** size(chunk) $\leq s$ **then**
11:       break
12:     $k \leftarrow k-1$
13:   subsetY $\leftarrow$ subsetY.add(chunk)
14:   **for** $y_j \in$ chunk **do**
15:     RemoveNode(graph,y_j)
16: **return** subsetY

ClusterY first creates a graph *graph* with $Y \setminus U$ as vertex set and edges between variables y_i and y_j that share a clause in $F(X,Y)$. ClusterY then calls subroutine kHopNeighbor for each variable y_i. The set of variables returned by kHopNeighbor is stored as chunk. If the size of chunk is greater than s, ClusterY reduces the value of k by one at line 12, and calls kHopNeighbor again with the updated value of k. Otherwise, ClusterY adds chunk to subsetY at line 13. Finally at line 15, ClusterY removes the nodes corresponding to each variable of chunk from *graph*.

3) CandidateSkF: Pseudocode for this routine is deferred to the technical report [22]. It takes a set Σ of samples, $F(X,Y)$, Ψ: a candidate function vector, chunk: the set of variables to learn candidates, and dependson: a partial dependency vector as input, and finds the candidates corresponding to each of the variables y_i in chunk. CandidateSkF assumes access to subroutines CreateDecisionTree and Path as described by Golia et al. [21]. The following are the additional subroutines used by CandidateSkF.

1) LeafNodes, which takes a decision tree dt as an input and returns a list of leaf nodes of dt.
2) Label(y_i, l), which takes a variable y_i and a leaf node l as input, and returns 1 if the class label corresponding to the node l has value 1 at the i^{th} index.

CandidateSkF starts off by initializing the set *featset* of features with the set X of input variables. It then attempts to find a list D of variables y_j such that $y_j \prec_d y_i$ where y_i belongs to chunk. Next, CandidateSkF adds $Y \setminus D$ to *featset*, and creates a decision tree dt using samples from Σ over *featset* to learn the chunk variables. For a leaf node l of dt, if Label(y_i, l) returns 1, then ψ_i is updated with the disjunction of the formula returned by subroutine Path. Finally, CandidateSkF iterates over all y_j occurring in ψ_i to add them to the list dependson[y_i].

4) FindRepairCandidates: Pseudocode for this routine is deferred to the technical report [22]. FindRepairCandidates starts with a LexMaxSAT call using hard constraints $F(X,Y) \wedge (X \leftrightarrow \sigma[X])$, soft constraints $(y_i \leftrightarrow \sigma[y_i'])$ for each y_i of Y. The preference order on soft constraints is given by *TotalOrder*. FindRepairCandidates calls the MaxSATList subroutine, which returns a list of Y variables *ind* such that the soft constraints corresponding to variables in *ind* were not satisfied by the optimal solution returned by the LexMaxSAT solver.

A. Example

We now illustrate our algorithm through an example.

Example 1: Let $X = \{x_1, x_2\}$, $Y = \{y_1, y_2, y_3, y_4\}$ in $\exists Y F(X,Y)$ where $F(X,Y)$ is $(x_1 \vee x_2 \vee y_1) \wedge (x_2 \vee \neg y_1 \vee y_2) \wedge (y_3 \vee y_4) \wedge (\neg y_3 \vee \neg y_4)$.

1) FindUniqueDef finds that y_4 is defined by $\{x_1, x_2, y_1, y_2, y_3\}$ and returns the Skolem function $\psi_4 = \neg y_3$. We get $Z = \{y_4\}$ as a determined set.
2) Next, Manthan2 generates training data through sampling (Figure 2). Manthan2 attempts to cluster $Y \setminus Z = \{y_1, y_2, y_3\}$ into different chunks of variables to learn together. As y_1 and y_2 share a clause, ClusterY returns the clusters $\{\{y_1, y_2\}, \{y_3\}\}$. Manthan2 now attempts to learn candidate Skolem functions ψ_1, ψ_2 together by creating a decision tree (Figure 3). The decision tree construction uses the samples of $\{x_1, x_2, y_3\}$ as features and samples of $\{y_1, y_2\}$ as labels. The candidate function ψ_1 is constructed by taking a disjunction over all paths that end in leaf nodes with label 1 at index 1 in the learned decision tree: as shown in Figure 3, ψ_1 is synthesised as $(x_1 \vee (\neg x_1 \wedge \neg x_2))$. Similarly, considering paths to leaf nodes with label 1 at index 2, we get $\psi_2 = (\neg x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_2)$, which simplifies to $\neg x_1$. Now, samples of $\{x_1, x_2, y_1, y_2\}$ are used to predict y_3. Considering the path to the leaf node of the learned decision tree with label 1, we get $\psi_3 = x_2$.
At the end of CandidateSkF, we have $\psi_1 := (x_1 \vee (\neg x_1 \wedge \neg x_2))$, $\psi_2 := \neg x_1$, $\psi_3 := x_2$, and $\psi_4 := \neg y_3$. Let us assume the total order returned by FindOrder is *TotalOrder* $= \{y_4, y_3, y_2, y_1\}$.
3) We construct the error formula, $E(X,Y,Y') = F(X,Y) \wedge \neg F(X,Y') \wedge (Y' \leftrightarrow \Psi)$, which turns out to be SAT with counterexample $\sigma = \langle x_1 \leftrightarrow 1,\ x_2 \leftrightarrow 0,\ y_1 \leftrightarrow 0,\ y_2 \leftrightarrow 1,\ y_3 \leftrightarrow 0,\ y_4 \leftrightarrow 1,\ y_1' \leftrightarrow 1,\ y_2' \leftrightarrow 0,\ y_3' \leftrightarrow 0,\ y_4' \leftrightarrow 1\rangle$.
FindRepairCandidates calls LexMaxSAT with $F(X,Y) \wedge (x_1 \leftrightarrow \sigma[x_1]) \wedge (x_2 \leftrightarrow \sigma[x_2])$ as hard constraints and $((y_1 \leftrightarrow \sigma[y_1']), 4) \wedge ((y_2 \leftrightarrow \sigma[y_2']), 3) \wedge ((y_3 \leftrightarrow \sigma[y_3']), 2) \wedge ((y_4 \leftrightarrow \sigma[y_4']), 1)$ as soft constraints, with the preference order of soft constraints indicated by their weights. FindRepairCandidates returns $ind = \{y_2\}$. Repair synthesis commences for ψ_2 with a satisfiability check of $G_2 = F(X,Y) \wedge (x_1 \leftrightarrow \sigma[x_1]) \wedge (x_2 \leftrightarrow \sigma[x_2]) \wedge (y_1 \leftrightarrow \sigma[y_1']) \wedge (y_2 \leftrightarrow \sigma[y_2'])$. The formula is unsatisfiable, and Manthan2 calls FindCore, which returns variable y_1, since the constraints $(y_1 \leftrightarrow \sigma[y_1'])$ and $(y_2 \leftrightarrow \sigma[y_2'])$ are not jointly satisfiable in G_2. As the output ψ_2 for the assignment σ must change from 0 to 1, ψ_2 is repaired by disjoining with y_1, and we get $\psi_2 := \neg x_1 \vee y_1$ as the new candidate. For the updated candidate vector Ψ the error formula is UNSAT, and thus Ψ is returned as a Skolem function vector.

x_1	x_2	y_1	y_2	y_3
0	0	1	1	0
0	1	0	1	1
1	1	1	0	1

Fig. 2: Samples of F

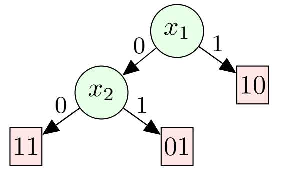


Fig. 3: Learned decision tree with labels $\{y_1, y_2\}$ and features $\{x_1, x_2, y_3\}$

V. Experimental Evaluation

We conducted an extensive study on 609 benchmarks that have been previously employed in studies [4], [5], [21]; in particular, we use instances from the 2QBF tracks of QBFEval'17 [1] and QBFEval'18 [2], and benchmarks related to arithmetic [53], disjunctive decomposition [6], and factorization [6]. We used Open-WBO [39] for unweighted MaxSAT queries, RC2 [27] for LexMaxSAT queries, and PicoSAT [10] to compute UnsatCore. Further, we used CryptoMiniSat [50] to find unates and a library based on UNIQUE [49] to extract unique Skolem functions. We used CMSGen [23] to sample the satisfying assignments of the specification. Finally, we used Scikit-Learn [3] to learn decision trees and ABC [36] to manipulate Boolean functions. 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 single-threaded mode on a single core with a timeout of 7200 seconds. We used the *PAR-2* score to compare different techniques, which corresponds to the Penalized Average Runtime, where for every unsolved instance there is a penalty of $2\times$ timeout.

The objective of our experimental evaluation was to compare the performance of Manthan2 with the state-of-the-art tools C2Syn [4], BFSS [5], CADET [42], and Manthan [21], and to analysis the impact of each of the algorithmic modifications implemented in Manthan2. In particular, our empirical evaluation sought answers to the following questions:

1) How does the performance of Manthan2 compare with state-of-the-art Skolem functional synthesis tools?
2) What is the impact on the performance of Manthan2 of each of the proposed modifications?

a) Summary of Results: Manthan2 outperforms all the state-of-the-art tools by solving 509 benchmarks, while the closest contender, Manthan [21] solves 356 benchmarks—an increase of **153** benchmarks over the state-of-the-art. It is worth emphasizing that the increment of 153 is more than twice the improvement shown by Manthan over CADET [42], which could solve 280 benchmarks.

Moreover, we found that extracting unique functions is useful. There are 246 benchmarks out of 609 for which the ratio of Y variables being uniquely defined to the total number of Y is greater than 95%, that is, Manthan2 could extract Skolem functions for that many variables via unique function extraction. There is an increase of 25 benchmarks in the number of solved instances by retaining variables in the determined set to learn and repair candidates. Further, learning candidate functions for a subset of variables together with the help of multi-classification reduces the PAR-2 score from 3227.11 to 2974.91. Finally, we see a reduction of 100 seconds in the PAR-2 score by LexMaxSAT.

TABLE I: Performance Summary over 609 benchmarks

	C2Syn	BFSS	CADET	Manthan	Manthan2
Solved	206	247	280	356	509
PAR-2	9594.83	8566.87	7817.58	6374.39	2858.61

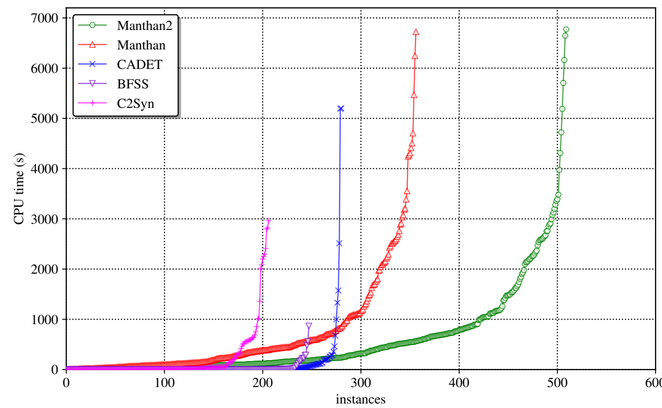


Fig. 4: Cactus plot: Manthan2 vis-a-vis state-of-the-art synthesis tools. Timeout 7200s. Total benchmarks: 609.

A. Manthan2 *vis-a-vis State-of-the-Art Synthesis Tools*

We compared Manthan2 with state-of-the-art tools: C2Syn [4], BFSS [5], CADET [42] and Manthan [21]. Figure 4 shows a cactus plot to compare the run-time performance of different synthesis tools. The x-axis represents the number of benchmarks and y-axis represents the time taken, a point $\langle x, y \rangle$ implies that a tool took less than or equal to y seconds to find a Skolem function vector for x many benchmarks out of total 609 benchmarks.

As shown in Figure 4, Manthan2 significantly improves on the state of the art techniques, both in terms of the number of instances solved and runtime performance. In particular, Manthan2 is able to solve 509 instances while Manthan can solve only 356 instances, thereby achieving an improvement of 153 instances in the number of instances solved. To measure the runtime performance in more detail, we computed PAR-2 scores for all the techniques. The PAR-2 scores for Manthan2 and Manthan are 2858.61 and 6374.39, which is an improvement of 3521.78 seconds. Finally, we sought to understand if Manthan2 performs better than the union of all the other tools. Here, we observe that Manthan2 solves 71 instances that the other tools could not solve, whereas there are only 40 instances not solved by Manthan2 that were solved by one of the other tools.

Manthan2 ***vis-a-vis*** Manthan*:* Table III presents a pairwise comparison of Manthan2 with Manthan. The first column (PreRepair) presents the number of benchmarks that needed no repair iteration to synthesise a Skolem function vector. The second column (Repair) represents the number of benchmarks that underwent repair iterations. The third column (Self-Sub) presents the number of benchmarks for which at least one variable underwent self-substitution.

TABLE II: Manthan2 vs. other state-of-the-art tools. *All tools* represents the union of all state-of-the-art tools.

		C2Syn	BFSS	CADET	Manthan	All
Manthan2	Less	17	18	21	24	40
	More	320	280	250	177	71

TABLE III: Pairwise comparison of Manthan2 with Manthan. The table represents the number of benchmarks solved with PreRepair, Repair, and Self-Substitution for Manthan and Manthan2.

	PreRepair	Repair	Self-Sub
Manthan	132	224	75
Manthan2	385	124	33

TABLE IV: Number of benchmarks with different maximum function size for uniquely defined variables. Function size is measured in terms of number of clauses.

	[1-10]	(10-100]	(100-1000]	(> 1000)
#-benchmarks	209	203	61	136

We investigate the reason for the increase in the number of benchmarks solved in PreRepair, and observed that Manthan2 could extract Skolem functions via unique function extraction for 90% of the variables for 274 out of these 385 benchmarks.

We also observed a significant decrease in the number of benchmarks that needed repair iterations. Out of 124 benchmarks that underwent repair to synthesise a Skolem function vector, only 33 benchmarks needed self-substitution with Manthan2, whereas there are 75 out of 224 benchmarks that needed self-substitution with Manthan. The fact that fewer benchmarks required self-substitution to synthesise a Skolem function vector shows that Manthan2 could find some hard-to-learn Skolem functions.

B. Performance Gain with Each Technical Contribution

1) Impact of Unique Function Extraction: We now present the impact of extracting Skolem function for uniquely defined variables. Figure 5 shows the percentage of uniquely determined functions on the x-axis, and number of benchmarks on y-axis. A bar at x shows that y many benchmarks had x% of Y variables that are uniquely defined. As shown in Figure 5, there are 246 benchmarks out of 609 with more than 95% uniquely defined variables; therefore, Manthan2 could extract Skolem functions corresponding to these variables via unique function extraction. There are only 5 benchmarks where all the Y variables are defined. Our analysis shows that extracting unique functions significantly reduces the number of Y variables that needed to be learned and repaired in the subsequent phases of Manthan2.

We also analyzed the performance of Manthan2 with respect to unique function size. Note that we measure size in terms of number of clauses, as the extracted functions are in CNF. A benchmark is considered to have size S if the maximum size among all its unique functions is S.

Table IV shows the number of benchmarks with different maximum unique function sizes. There are 136 benchmarks for which at least one uniquely defined variable has function size greater than 1000 clauses. In general, larger size functions require more data to learn. Table IV shows that Manthan2 was able to extract some hard-to-learn Skolem functions.

An interesting observation is that there were 54 benchmarks that required self-substitution for just one variable with Manthan. However, Manthan2 was able to identify that particular variable as uniquely defined and the corresponding function size was more than 3000 clauses. This observation emphasizes that it is important to extract the functions for uniquely defined variables with large function size in order to efficiently synthesise a Skolem function vector. Therefore, even if there is only one variable with large function size, it is important to extract the corresponding function—the reason for considering maximum size instead of mean or median size in Table IV.

2) Impact of Learning and Repairing over Determined Features: We now present the impact of variable retention. Manthan2 could solved 502 instances with a PAR-2 score of 3227.11 by retaining variables in the *determined set* to use them further as features in learning and repairing the other candidates, whereas, if we eliminate them, it could solve only 477 instances with a PAR-2 score of 3523.28—a difference of 25 benchmarks.

It is worth mentioning that there are 370 instances that needed no repair iterations (solved in PreRepair) to synthesise a Skolem function vector when learned with determined features, whereas, if Manthan2 does not consider determined features, we see a reduction of 6 benchmark in the number of instances solved in PreRepair.

Interestingly, even if we have fewer such determined features, it is essential to use them to learn and repair the candidates. For example, considering the benchmark *query64_01*, there are only five variables out of 597 total Y variables that could be identified as determined features. If we eliminate those five variables, Manthan2 could not synthesise a Skolem function vector even with more than 150 repair iterations within a timeout of 7200s. However, if we retain them as determined features, Manthan2 could synthesise a Skolem function vector within 9 repair iterations in less than 400s.

3) Efficacy of Multi-Classification and Impact of LexMaxSAT: As discussed in Section III, two essential questions arise when using multi-classification to learn candidates for a subset of Y together: 1) how to divide the Y variables into different subsets, and 2) how many variables should be learned together?

We experimented with following techniques to divide Y variables into subsets of sizes 5 and 8, i.e, s = 5 or 8:

1) Randomly dividing Y variables into different disjoint subsets.
2) Clustering Y variables in accordance to the edge distance (parameter k) in the primal graph: (i) using $k = 2$ (ii) using $k = 3$

Figure 6 shows a heatmap of PAR-2 scores for different configurations of Manthan2. A lower PAR-2 score, i.e., a tilt towards the red end of the spectrum in Figure 6, indicates a favorable configuration. The columns of Figure 6 correspond to different ways of dividing Y variables into different subsets: (i) *Random*, (ii) $k = 2$, and (iii) $k = 3$. The rows of Figure 6 show results for different maximum sizes of such subsets, i.e., s = 5, 8. The number of instances solved in each configuration is also shown in brackets. For comparison, the PAR-2 score of Manthan2 with binary classification is 3227.11s and it solved 502 benchmarks.

Let us first discuss Figure 6a, i.e, the results without LexMaxSAT. Manthan2 shows a performance improvement with the proposed clustering-based approach in comparison to randomly dividing Y variables into subsets. As shown in Figure 6a, we observed a drop in PAR-2 score when moving from random to cluster-based partitioning of Y variables.

We see a better PAR-2 score with graph-based multi-classification compared to binary classification, though the number of instances solved (except with k=3, s=5) is lower than the number of instances solved with binary classification. This shows that dividing Y variables using a cluster-based approach is effective in reducing the candidate learning time. Manthan2 performs best with $k = 3$ and $s = 5$, where it could solve 503 benchmarks (1 more instance than with binary classification) with a PAR-2 score of 2974.9s, which amounts to a reduction of 252 seconds over the PAR-2 score with binary

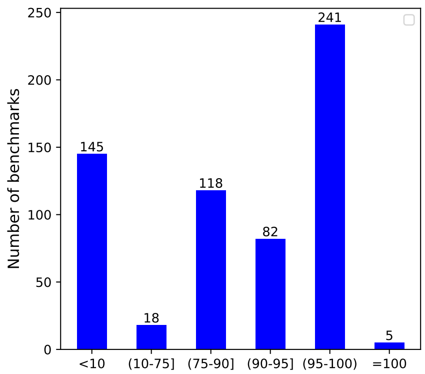
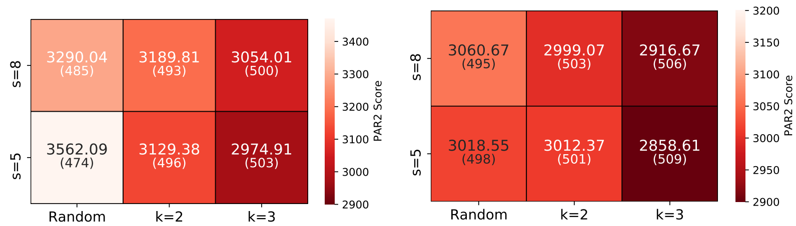


Fig. 5: Number of benchmarks by % ratio of uniquely defined output variables for all 609 benchmarks.



(a) LexMaxSAT turned off. **(b)** LexMaxSAT turned on.

Fig. 6: Heatmap of PAR-2 scores achieved by different configuration of Manthan2 (darker is better). Here, s represents the *size* of sets of variables that were learned together, and $k = 2, k = 3$ represents the edge distance in the primal graph used to cluster output variables. The number of instances solved by each configuration is shown in brackets. [Best viewed in color].

classification. We observe a similar trend with LexMaxSAT turned on (as shown in Figure 6b).

Finally, let us move our attention towards the impact of LexMaxSAT, shown in Figure 6b. Manthan2 uses LexMaxSAT only if the number of candidates to repair exceeds 50 times the number of candidates chosen by MaxSAT. A comparison of Figure 6a and Figure 6b shows that with LexMaxSAT, Manthan2 solves at least 3 more benchmarks for all the configurations.

Manthan2 performs best when we turn on LexMaxSAT and set $k = 3$ as well as $s = 5$. The results discussed in Section V-A were achieved with this configuration.

VI. Related Work

Boolean functional synthesis is a classical problem. Its origin traces back to Boole's seminal work [12], which was subsequently pursued with a focus on decidability—by Löwenheim and Skolem [37].

The past decade has seen significant progress in the development of efficient tools for Boolean functional synthesis, driven by a diverse set of techniques. Quantifier elimination by functional composition can be an efficient approach when paired with Craig interpolation to reduce the size of composite functions [29], [30]. However, interpolation does not reliably find succinct composite functions, thus limiting scalability of this method. More recently, it was shown that ROBDDs lend themselves well to functional composition [17] (even without interpolation) and furthermore, they can take advantage of factored specifications [53].

Instead of directly deducing Skolem functions from a specification, a series of CEGAR-based synthesis algorithms start from an initial set of approximate functions that are rectified in a subsequent phase of counterexample guided refinement [5], [6], [32]. It was observed that the initial functions are often valid Skolem functions [5]. This naturally leads to the question as to which classes of specifications admit efficient Boolean functional synthesis, which has recently been studied from the area of knowledge compilation [4], [5].

So-called incremental determinization can be seen as lifting Conflict-Driven Clause Learning (CDCL) to the level of Boolean functions [42], [43], [45]: variables with unique Skolem functions are successively identified, in analogy with unit propagation, and whenever this process comes to a halt, a Skolem function for one of the remaining variables is fixed by adding auxiliary clauses. While originally developed as a decision procedure for 2QBF, the algorithm was later successfully adapted to perform functional synthesis for non-valid specifications [42].

Skolem functions can also be efficiently extracted from proofs generated by QBF solvers [8], [9], [26], [41], [44], [48], but this requires both a valid input specification and a proof of validity (which itself is typically hard to compute).

Recently, a data-driven approach to Boolean functional synthesis was proposed [21]. Data-driven approaches have proven to be efficient for the other forms of synthesis, like invariant synthesis [15], [19], [25], or synthesis by example [16].

Our data-driven approach benefits from identifying variables that are defined by a subset of input variables, since the corresponding definitions represent Skolem functions that do not have to be learned. Such definitions are often introduced as an artifact of converting circuits into CNF formulas, where gates are encoded by auxiliary variables that are defined in term of their inputs. Standard techniques for recovering gate definitions from CNF formulas (these are also used in Boolean synthesis tools [4], [5]) rely on pattern matching of clauses and variables induced by specific gate types [18], [24], [46]. These methods are fast but can only detect definitions from a predefined library of gates. By contrast, Manthan2 extracts the functions for uniquely defined variables using semantic gate extraction based on propositional interpolation [49]. This approach is computationally more expensive (each definability check requires a SAT call), but it is complete: whenever a variable y is defined in terms of a given set X of variables, the corresponding definition will be returned.

VII. Conclusion

In this paper, we showed how to improve the state-of-the-art data-driven Skolem function synthesiser Manthan to achieve better scalability. We proposed crucial algorithm innovation, and used them in a new framework, called Manthan2. Manthan2 could synthesise a Skolem function vector for 509 instances out a total of 609, compared to 356 instances solved by Manthan.

Acknowledgments: 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], NUS ODPRT Grant [R-252-000-685-13], and the Vienna Science and Technology Fund (WWTF) [ICT19-060]. The computational work for this article was performed on resources of the National Supercomputing Center, Singapore: https://www.nscc.sg.

References

[1] "QBF solver evaluation portal 2017." [Online]. Available: http://www.qbflib.org/qbfeval17.php

[2] "QBF solver evaluation portal 2018." [Online]. Available: http://www.qbflib.org/qbfeval18.php

[3] "sklearn.tree.decisiontreeclassifier." [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html

[4] S. Akshay, J. Arora, S. Chakraborty, S. Krishna, D. Raghunathan, and S. Shah, "Knowledge compilation for Boolean functional synthesis," in *Proc. of FMCAD*, 2019.

[5] S. Akshay, S. Chakraborty, S. Goel, S. Kulal, and S. Shah, "What's hard about Boolean functional synthesis?" in *Proc. of CAV*, 2018.

[6] S. Akshay, S. Chakraborty, A. K. John, and S. Shah, "Towards parallel Boolean functional synthesis," in *Proc. of TACAS*, 2017.

[7] C. Ansótegui, M. L. Bonet, J. Gabas, and J. Levy, "Improving wpm2 for (weighted) partial maxSAT," in *Proc. of CP*, 2013.

[8] V. Balabanov and J.-H. R. Jiang, "Unified QBF certification and its applications," in *Proc. of FMCAD*, 2012.

[9] V. Balabanov, J. R. Jiang, M. Janota, and M. Widl, "Efficient extraction of QBF (counter)models from long-distance resolution proofs," in *Proc. of AAAI*, 2015.

[10] A. Biere, "PicoSAT essentials," *Proc. of JSAT*, 2008.

[11] A. Biere, F. Lonsing, and M. Seidl, "Blocked clause elimination for QBF," in *Proc. of CADE*, 2011.

[12] G. Boole, *The mathematical analysis of logic*. Philosophical Library, 1847.

[13] R. K. Brayton, "Boolean relations and the incomplete specification of logic networks," in *Proc. of VLSID*, 1989.

[14] R. K. Brayton and F. Somenzi, "An exact minimizer for boolean relations," in *Proc. of ICCAD*, 1989.

[15] P. Ezudheen, D. Neider, D. D'Souza, P. Garg, and P. Madhusudan, "Horn-ICE learning for synthesizing invariants and contracts," in *Proc. of OOPSLA*, 2018.

[16] G. Fedyukovich and A. Gupta, "Functional synthesis with examples," in *Proc. of CP*, 2019.

[17] D. Fried, L. M. Tabajara, and M. Y. Vardi, "BDD-based Boolean functional synthesis," in *Proc. of CAV*, 2016.

[18] Z. Fu and S. Malik, "Extracting logic circuit structure from conjunctive normal form descriptions," in *Proc. of VLSID*, 2007.

[19] P. Garg, C. Löding, P. Madhusudan, and D. Neider, "ICE: A robust framework for learning invariants," in *Proc. of CAV*, 2014.

[20] A. Ghosh, S. Devadas, and A. R. Newton, "Heuristic minimization of boolean relations using testing techniques," *IEEE transactions on computer-aided design of integrated circuits and systems*, 1992.

[21] P. Golia, S. Roy, and K. S. Meel, "Manthan: A data driven approach for Boolean function synthesis," in *Proc. of CAV*, 2020.

[22] P. Golia, F. Slivovsky, S. Roy, and K. S. Meel, "Engineering an efficient Boolean functional synthesis engine," 2021. [Online]. Available: https://arxiv.org/abs/2108.05717

[23] P. Golia, M. Soos, S. Chakraborty, and K. S. Meel, "Designing samplers is easy: The boon of testers," in *Proc. of FMCAD*, 2021.

[24] A. Goultiaeva and F. Bacchus, "Recovering and utilizing partial duality in QBF," in *Proc. of SAT*, 2013.

[25] O. Grumberg, F. Lerda, O. Strichman, and M. Theobald, "Proof-guided underapproximation-widening for multi-process systems," in *Proc. of POPL*, 2005.

[26] M. J. Heule, M. Seidl, and A. Biere, "Efficient extraction of Skolem functions from QRAT proofs," in *Proc. of FMCAD*, 2014.

[27] A. Ignatiev, A. Morgado, and J. Marques-Silva, "PySAT: A Python toolkit for prototyping with SAT oracles," in *Proc. of SAT*, 2018.

[28] S.-W. Jeong and F. Somenzi, "A new algorithm for the binate covering problem and its application to the minimization of boolean relations," in *Proc. of ICCAD*, 1992.

[29] J.-H. R. Jiang, "Quantifier elimination via functional composition," in *Proc. of CAV*, 2009.

[30] J. R. Jiang, H. Lin, and W. Hung, "Interpolating functions from large boolean relations," in *Proc. of ICCAD*. ACM, 2009, pp. 779–784.

[31] S. Jo, T. Matsumoto, and M. Fujita, "SAT-based automatic rectification and debugging of combinational circuits with lut insertions," *Proc. of IPSJ T-SLDM*, 2014.

[32] A. K. John, S. Shah, S. Chakraborty, A. Trivedi, and S. Akshay, "Skolem functions for factored formulas," in *Proc. of FMCAD*, 2015.

[33] J. H. Kukula and T. R. Shiple, "Building circuits from relations," in *Proc. of CAV*, 2000.

[34] J. Lang and P. Marquis, "On propositional definability," *Artificial Intelligence*, 2008.

[35] B. Lin and F. Somenzi, "Minimization of symbolic relations," in *Proc. of ICCAD*, 1990.

[36] B. Logic and V. Group, "ABC: A system for sequential synthesis and verification." [Online]. Available: http://www.eecs.berkeley.edu/~alanmi/abc/

[37] L. Löwenheim, "Über die Auflösung von Gleichungen im logischen Gebietekalkul," *Mathematische Annalen*, 1910.

[38] J. Marques-Silva, J. Argelich, A. Graça, and I. Lynce, "Boolean lexicographic optimization: algorithms & applications," *Proc. of Annals of Mathematics and Artificial Intelligence*, 2011.

[39] R. Martins, V. Manquinho, and I. Lynce, "Open-WBO: A modular MaxSAT solver," in *Proc. of SAT*, 2014.

[40] F. Massacci and L. Marraro, "Logical cryptanalysis as a SAT problem," *Journal of Automated Reasoning*, 2000.

[41] A. Niemetz, M. Preiner, F. Lonsing, M. Seidl, and A. Biere, "Resolution-based certificate extraction for QBF," in *Proc. of SAT*, 2012.

[42] M. N. Rabe, "Incremental determinization for quantifier elimination and functional synthesis," in *Proc. of CAV*, 2019.

[43] M. N. Rabe and S. A. Seshia, "Incremental determinization," in *Proc. of SAT*, 2016.

[44] M. N. Rabe and L. Tentrup, "CAQE: A certifying QBF solver," in *Proc. of FMCAD*, 2015.

[45] M. N. Rabe, L. Tentrup, C. Rasmussen, and S. A. Seshia, "Understanding and extending incremental determinization for 2QBF," in *Proc. of CAV*, 2018.

[46] J. A. Roy, I. L. Markov, and V. Bertacco, "Restoring circuit structure from SAT instances," in *Proc. of IWLS*, 2004.

[47] M. Samer and S. Szeider, "Algorithms for propositional model counting," *Journal of Discrete Algorithms*, 2010.

[48] M. Schlaipfer, F. Slivovsky, G. Weissenbacher, and F. Zuleger, "Multi-linear strategy extraction for QBF expansion proofs via local soundness," in *Proc. of SAT*, 2020.

[49] F. Slivovsky, "Interpolation-based semantic gate extraction and its applications to QBF preprocessing," in *Proc. of CAV*, 2020.

[50] M. Soos, "msoos/cryptominisat," 2019. [Online]. Available: https://github.com/msoos/cryptominisat

[51] M. Soos, S. Gocht, and K. S. Meel, "Tinted, detached, and lazy CNF-XOR solving and its applications to counting and sampling," in *Proc. of CAV*, 2020.

[52] S. Srivastava, S. Gulwani, and J. S. Foster, "Template-based program verification and program synthesis," *STTT*, 2013.

[53] L. M. Tabajara and M. Y. Vardi, "Factored Boolean functional synthesis," in *Proc. of FMCAD*, 2017.