Synthesis with Explicit Dependencies

Priyanka Golia
Indian Institute of Technology Kanpur
National University of Singapore

Subhajit Roy
Indian Institute of Technology Kanpur

Kuldeep S. Meel
National University of Singapore

***Abstract*—Quantified Boolean Formulas (QBF) extend propositional logic with quantification $\forall, \exists$. In QBF, an existentially quantified variable is allowed to depend on all universally quantified variables in its scope. Dependency Quantified Boolean Formulas (DQBF) restrict the dependencies of existentially quantified variables. In DQBF, existentially quantified variables have explicit dependencies on a subset of universally quantified variables, called Henkin dependencies. Given a Boolean specification between the set of inputs and outputs, the problem of Henkin synthesis is to synthesize each output variable as a function of its Henkin dependencies such that the specification is met. Henkin synthesis has wide-ranging applications, including verification of partial circuits, controller synthesis, and circuit realizability.**

This work proposes a data-driven approach for Henkin synthesis called Manthan3. On an extensive evaluation of over 563 instances arising from past DQBF solving competitions, we demonstrate that Manthan3 is competitive with state-of-the-art tools. Furthermore, Manthan3 solves 26 benchmarks that none of the current state-of-the-art techniques could solve.

I. Introduction

Quantified Boolean Formulas (QBF) equip the propositional logic with universal ($\forall$) and existential quantifiers ($\exists$) for propositional variables. In QBF, an existentially quantified variable is allowed to depend on all universally quantified variables within its scope. On the other hand, Henkin quantifiers, often called Branching quantifiers, generalize the standard quantification and allow explicit declarations of dependencies [20]. Propositional logic is equipped with Henkin quantifiers, resulting in the so-called Dependency Quantified Boolean Formulas (DQBFs). In DQBF, an existentially quantified variable is allowed to depend on a pre-defined subset of universally quantified variables, called Henkin dependencies. For example, $\phi : \forall x_1, x_2 \exists^{x_1} y_1.\varphi(x_1, x_2, y_1)$ is a DQBF formula, where φ is some quantifier-free Boolean formula and existentially quantified variable y_1 is only allowed to depend on x_1, which is the Henkin dependency corresponding to y_1. The dependency specification quantification is called Henkin quantifier [20].

These explicit dependencies provide more succinct descriptive power to DQBF than QBF. However, DQBF is shown to be in the complexity class of NEXPTIME-complete [24], whereas QBF is *only* PSPACE-complete [9]. The payoffs associated with an increase in the computational complexity are the wide-ranging applications of DQBF, such as engineering change of order [19], topologically constrained synthesis [3], equivalence checking of partial functions [10], finding strategies for incomplete games [24], controller synthesis [5], circuit realizability [3].

The DQBF satisfiability is a decision problem that looks for an answer to the question: *Does there exist a function corresponding to each existentially quantified variable, in terms of its Henkin dependencies, such that the formula substituted with the function in places of existentially quantified variables is a tautology?* Owing to wide variety of applications that can be represented as DQBF, recent years have seen a surge of interest in DQBF solving [8], [10], [26], [29], [30]. In many cases, a mere True/False answer is not sufficient as one is often interested in determining the definitions corresponding to those functions. For instance, in the context of engineering change of order (ECO), in addition to just knowing whether the given circuit could be rectified to meet the *golden* specification, one would also be interested in deriving corresponding patch functions [19]. Owing to the naming of dependencies, we call such patch functions to be Henkin functions.

Recent years have witnessed an increased interest in the problem of Henkin function synthesis. The current state-of-the-art techniques, HQS2 [30] and Pedant [26] can synthesize Henkin functions for True DQBF in addition to DQBF solving. HQS2 [31] applies a sequence of transformations to eliminate quantifiers in DQBF instances to synthesize Henkin functions for True instances, whereas Pedant [26] uses interpolation-based definition extraction and various SAT oracle calls to synthesize Henkin functions. Despite the significant progress over the years, many real-world instances are beyond the reach of Henkin function synthesis engines.

In this work, we take a step to push the envelope of Henkin synthesis. To this end, we propose a novel framework for Henkin function synthesis, called Manthan3. Manthan3 takes an orthogonal approach to the existing techniques by combining advances in machine learning with automated reasoning. In particular, Manthan3 uses constrained sampling to generate the data, which is later fed to a machine-learning algorithm to learn the candidate functions in accordance with the Henkin dependencies for each existentially quantified variable. Then, Manthan3 employs a SAT solver to check the correctness of the synthesized candidates. If the candidate verification checks fail, Manthan3 does a counterexample-driven candidate repair. Furthermore, Manthan3 utilizes a MaxSAT solver-based method to find the candidates that need to undergo repair and uses a proof-guided strategy to construct a *good* repair.

To demonstrate the practical efficiency of Manthan3, we perform an extensive comparison with the prior state-of-the-art techniques, HQS2 and Pedant, over a benchmark suite of 563 instances. Our empirical evaluation demonstrates that Manthan3 shows competitive performance and significantly contributes to the portfolio of Henkin synthesizers. Manthan3 achieves

978-3-9819263-7-8/DATE23/© 2023 EDAA

the shortest synthesizing time on 42 of the 204 benchmarks solved by at least one tool. Furthermore, Manthan3 is able to synthesize Henkin functions for 26 instances that none of the state-of-the-art function synthesis engines could synthesize.

II. Preliminaries

We use a lower case letter to represent a propositional variable and an upper case letter to represent a set of variables. A literal is either a variable or its negation, and a clause is considered as a disjunction of literals. A formula φ represented as conjunction of clauses is considered in Conjunctive Normal Form (CNF). *Vars*(φ) represents the set of variables appearing in φ. A satisfying assignment(σ) of the formula φ maps *Vars*(φ) to $\{0,1\}$ such that φ evaluates to True under σ. We use $\sigma \models \varphi$ to represent σ as a satisfying assignment of φ. For a set of variables V, we used $\sigma[V]$ to denote the restriction of σ to V. If φ evaluates to True for all possible valuation of *Vars*(φ), φ is considered as tautology.

A uniform sampler samples the required number of satisfying assignments uniformly at random from the solution space of the formula. We use UnsatCore to represents an *unsatisfiable core*, which is a subset of clauses of φ for which there does not exists a satisfying assignment. For a CNF formula in which a set of clauses is considered as hard constraints and remaining clauses as soft constraints, a MaxSAT solver tries to find a satisfying assignment that satisfies all hard constraints and maximizes the number of satisfied soft constraints.

A formula ϕ is DQBF if it can be represented as $\phi : \forall x_1 \dots x_n \; \exists^{H_1} y_1 \dots \exists^{H_m} y_m \varphi(X,Y)$ where $X = \{x_1,\dots,x_n\}$, $Y = \{y_1,\dots,y_m\}$ and $H_i \subseteq X$ represents the dependency set of y_i, that is, variable y_i can only depend on H_i. Each H_i is called Henkin dependency and each quantifier $\exists^{H_i}$ is called *Henkin* quantifier [17].

A DQBF ϕ is considered to be True, if there exists a function $f_i : \{0,1\}^{|H_i|} \mapsto \{0,1\}$ for each existentially quantified variable y_i, such that $\varphi(X, f_1(H_1), \dots, f_m(H_m))$, obtained by substitution of each y_i by its corresponding function f_i, is a tautology. Given a DQBF ϕ, the problem of DQBF satisfiability, is to determine whether a given DQBF is True or False.

Problem Statement: Given a True DQBF $\forall x_1 \dots x_n$ $\exists^{H_1} y_1 \dots \exists^{H_m} y_m \varphi(x_1,\dots,x_n,y_1,\dots,y_m)$ where $x_1,\dots,x_n \in X$, $y_1,\dots,y_m \in Y$, $H_i \subseteq X$, the problem of **Henkin Synthesis** is to synthesize a function vector $\boldsymbol{f} : \langle f_1,\dots,f_m \rangle$ such that $\varphi(X, f_1(H_1), \dots, f_m(H_m))$ is a tautology.

$\boldsymbol{f}$ is called Henkin function vector and each f_i is a Henkin function. We used $\forall X \exists^{H_1} y_1 \dots \exists^{H_m} y_m \; \varphi(x_1,\dots,x_n, y_1,\dots,y_m)$ and $\forall X \exists^{H} Y \varphi(X,Y)$ interchangeably.

Henkin synthesis generalizes Skolem synthesis in which $H_1 = \dots = H_m = X$. In such a case, one omits the usage of H_i and simply represents ϕ as $\forall X \exists Y \varphi(X,Y)$. In such a case, $\boldsymbol{f}$ is called Skolem function vector, such that $\forall X (\exists Y \varphi(X,Y) \leftrightarrow \varphi(X,\boldsymbol{f}))$.

III. Overview

This section provides a high-level overview of Manthan3 framework. While Manthan3 shares high-level similarity with Manthan, the recently proposed Skolem function synthesis engine [12] [14], in its usage of machine learning techniques and SAT/MaxSAT solvers, the two techniques differ crucially due to the requirements imposed by Henkin dependencies. It is worth remarking that handling Henkin dependencies is not trivial, perhaps best highlighted by the fact that 2-QBF is Σ_2^P-complete while DQBF is NEXPTIME-complete [24].

Manthan3 first uses advances in constrained sampling to generate the data, then use the data to learn a candidate vector $\boldsymbol{f}$ using a machine learning-based approach. Then, Manthan3 attempts to verify if the candidate vector $\boldsymbol{f}$ is a Henkin function vector. If the candidates pass the formal verification check, Manthan3 returns the candidates as a valid Henkin vector. Otherwise, the candidate vector is repaired to satisfy the counterexample, and the verification check is repeated. Note that Manthan3 needs to take care of restrictions imposed by Henkin dependencies while learning and repairing the candidates.

We now present high-level overview of Manthan3:

Data Generation: As the first step, Manthan3 uses constrained samplers [15], [16] to sample the satisfying assignments of specification φ uniformly at random from the solution space of specification. The sampled satisfying assignments are considered data to feed the learning algorithms to learn candidate functions in the next stage.

Candidate Learning: Manthan3 learns a binary decision tree classifier for each existentially quantified variable y_i to learn the candidate function f_i corresponding to it. The valuations of y_i in the generated samples are considered labels, and the valuations of corresponding Henkin dependencies H_i are considered the feature set to learn a decision tree. A Henkin function f_i corresponding to y_i is computed as a disjunction of labels along all the paths from the root node to leaf nodes with label **1** in the learned decision tree.

Due to the Henkin dependencies, the feature set for y_i must be restricted only to H_i. However, in order to learn a good decision tree, we can include all the y_j in the set of features for which $H_j \subset H_i$. The function f_j can be simply expanded within f_i so that f_i is only expressed in terms of H_i. For the cases when $H_j = H_i$, such use of the Y variables is allowed as long it does not cause the cyclic dependencies; that is, if y_j appears in the learned candidate f_i, then y_i is not allowed as a feature to learn candidate f_j. If y_j appears in f_i, then we say y_i depends on y_j, denoted as $y_i \prec_d y_j$. Manthan3 discovers requisite variable ordering constraints among such Y variables on the fly as the candidate functions are learned.

A function vector $\boldsymbol{f}$ in which y_j appears in f_i is a valid vector if y_i does not appear in f_j. If $\boldsymbol{f}$ is a valid function vector, there exists a partial order $\prec_d$ over $\{y_1,\dots,y_m\}$. Once, we have a candidate vector, Manthan3 obtains a valid linear extension total order, say denoted as Order, from the partial dependencies learned in *candidate learning* over Y variables.

Verification: The learned candidate vector may not always be a valid Henkin vector. Therefore, the candidate functions must be verified. $\boldsymbol{f}$ is a Henkin function vector only if $\varphi(X, f_1(H_1),$ $\dots, f_m(H_m))$ is a tautology. Manthan3 first, make a SAT oracle query on the formula $E(X,Y') = \neg\varphi(X,Y') \wedge (Y' \leftrightarrow \boldsymbol{f})$

If formula $E(X,Y')$ is UNSAT, Manthan3 returns the function vector $\boldsymbol{f}$ as a Henkin function vector. If formula $E(X, Y')$ is SAT and δ is a satisfying assignment of $E(X, Y')$, Manthan3 needs to find out whether $\varphi(X, Y)$ has a propositional model extending assignment of X. Therefore, Manthan3 performs another satisfiability check on formula $\varphi(X, Y) \wedge (X \leftrightarrow \delta[X])$. If satisfiability checks return UNSAT, the corresponding $DQBF$ formula is False, and there does not exist a Henkin function vector; therefore, Manthan3 terminates. Furthermore, if $\varphi(X, Y) \wedge (X \leftrightarrow \delta[X])$ is SAT, and π is a satisfying assignment and we need to repair the candidate function vector. Note that $\pi[X]$ is same as $\delta[X]$, and $\pi[Y]$ is a possible extending assignment of X, and $\delta[Y']$ presents the output of candidate function vector with $\delta[X]$. Now, we have a counterexample σ as $\pi[X] + \pi[Y] + \delta[Y']$.

Candidate Repair: We apply a counterexample driven repair approach for candidate functions. As Manthan3 attempts to fix the counterexample σ, it first needs to find which candidates to repair out of f_1 to f_m candidates. Manthan3 takes help of MaxSAT solver to find out the repair candidates, and makes a MaxSAT query with $\varphi(X, Y) \wedge (X \leftrightarrow \sigma[X])$ as hard constraints and $(Y \leftrightarrow \sigma[Y'])$ as soft constraints. It selects a function f_i for repair if the corresponding soft constraint $y_i \leftrightarrow \sigma[y_i']$ is falsified in the solution returned by the MaxSAT solver. Once, we have candidates to repair, Manthan3 employs unsatisfiability cores obtained from the infeasibility proofs capturing the reason for candidates to not meet the specification to construct a repair.

Let us now assume that Manthan3 selects f_i corresponding to variable y_i as a potential candidate. Manthan3 constructs another formula $G_i(X, Y)$ (Formula 1) to find the repair:

$$G_i(X, Y) : \varphi(X, Y) \wedge (H_i \cup \hat{Y} \leftrightarrow \sigma[H_i \cup \hat{Y}]) \wedge (y_i \leftrightarrow \sigma[y_i'])$$
$$\text{where } \hat{Y} \subseteq Y \text{ such that } \forall y_j \in \hat{Y} : H_j \subseteq H_i$$
$$\text{and } \{\mathsf{Order}[index(y_j)] > \mathsf{Order}[index(y_i)]\} \quad (1)$$

Informally, in order to determine whether f_i needs to be repaired, we conjunct the specification $\varphi(X, Y)$ with the conjunction of unit clauses that set the valuation of y_i to the current output of f_i and the valuation of all the dependencies as per the counter-example. We describe the intuition behind construction of $G_i(X, Y)$. The formula $G_i(X, Y)$ is constructed to answer the following question: *Whether is it possible for y_i to be set to the output of f_i given the valuation of its Henkin dependencies?*.

The answer to the above question depends on whether $G_i(X, Y)$ is UNSAT or SAT. $G_i(X, Y)$ being UNSAT indicates that it is not possible for y_i to be set to the output of f_i and the UnsatCore of $G_i(X, Y)$ captures the reason. Accordingly, Manthan3 uses the UnsatCore of $G_i(X, Y)$ to repair the candidate function f_i. In particular, Manthan3 uses all the variables corresponding to unit clauses in UnsatCore of $G_i(X, Y)$ to construct a repair formula β, and depending on the valuation of y_i' in the counter example σ, β is used to strengthen or weaken the candidate f_i to satisfy the counterexample.

On the other hand, if $G_i(X, Y)$ is SAT, Manthan3 attempts to find alternative candidate functions to repair. $G_i(X, Y)$ being SAT indicates that with the current valuation to Henkin dependencies, y_i could take a value as per the output of candidate f_i; however, to fix the counterexample σ, we need to repair another candidate function. To this end, let ρ be a satisfying assignment of $G_i(X, Y)$, then all y_j variables for which $\rho[y_j]$ is not the same as $\sigma[y_j']$ are added to the queue of potential candidates to repair.

The repair loop continues until either $E(X, Y')$ is UNSAT or $\varphi(X, Y) \wedge (X \leftrightarrow \delta[X])$ is UNSAT, where δ is a satisfying assignment of $E(X, Y')$. If $E(X, Y')$ is UNSAT, we have a Henkin function vector $\boldsymbol{f}$, and if $\varphi(X, Y) \wedge (X \leftrightarrow \delta[X])$ is UNSAT, then the given DQBF instance is False and there does not exist a Henkin function vector.

Algorithm 1 Manthan3($\forall X \exists^H Y.\varphi(X, Y)$)

$\Sigma \leftarrow$ GetSamples($\varphi(X, Y)$)
$D \leftarrow \{d_1 = \emptyset \ldots, d_{|Y|} = \emptyset\}$
for $\langle H_i, H_j \rangle$ **do**
  if $H_j \subset H_i$ **then**
    $d_j \leftarrow d_j \cup y_i$
for $y_i \in Y$ **do**
  $f_i, D \leftarrow$CandidateHkF($\Sigma, \varphi(X, Y), y_i, D$)
Order $\leftarrow$ FindOrder(D)
repeat
  $E(X, Y') \leftarrow \neg\varphi(X, Y') \wedge (Y' \leftrightarrow \boldsymbol{f})$
  $ret, \delta \leftarrow$ CheckSat($E(X, Y')$)
  if ret = SAT **then**
    $res, \pi \leftarrow$ CheckSat($\varphi(X, Y) \wedge (X \leftrightarrow \delta[X])$)
    if res = UNSAT **then**
      return $\forall X \exists^H Y.\varphi(X, Y)$ is False.
    $\sigma \leftarrow \pi[X] + \pi[Y] + \delta[Y']$ {σ is a counterexample}
    $\boldsymbol{f} \leftarrow$ RepairHkF($\varphi(X, Y), \boldsymbol{f}, \sigma$, Order)
until ret = UNSAT
$\boldsymbol{f} \leftarrow$ Substitute($\varphi(X, Y), \boldsymbol{f}$, Order)
return $\boldsymbol{f}$

IV. Algorithmic Details

Manthan3 (Algorithm 1) takes a DQBF instance $\forall X \exists^{H_1} y_1 \ldots \exists^{H_m} y_m \varphi(X, Y)$ as input and outputs a Henkin function vector $\boldsymbol{f} := \langle f_1, \ldots, f_m \rangle$.

Algorithm 1 assumes access to the following subroutines:

1) GetSamples: It takes a specification as input and calls an oracle to produce samples Σ of specifications. Each sample in Σ is a satisfying assignment of specifications.
2) CandidateHkF: This subroutine generates the candidate function corresponding to an existential variable. It takes a specification φ, generated samples Σ, existential variable y_i corresponding to which we want to learn a candidate function and a vector D that keeps track of dependencies among Y variables as input. CandidateHkF returns a candidate function f_i corresponding to y_i, and updates the dependencies in D for y_i. We discussed CandidateHkF routine in detail in Algorithm 2.
3) FindOrder: It takes a set D collection of d_i, where each d_i is the list of Y variables, which can depend on y_i. FindOrder obtains a valid linear extension, Order, from the partial dependencies in D.
4) CheckSat: It takes a specification as input and makes a SAT oracle call to do a satisfiability check on the specification. It returns the outcome of satisfiability check as SAT or UNSAT. In the case of SAT, it also returns a satisfiable assignment of the specification.

Algorithm 2 CandidateHkF$(\Sigma, \varphi(X,Y), y_i, D)$

1: $featset \leftarrow H_i$
2: **for** $y_j \in Y$ **do**
3:   **if** $(H_j \subseteq H_i) \wedge (y_j \notin (d_i \cup y_i))$ **then**
4:     $featset \leftarrow featset \cup y_j$
5: $feat, lbl \leftarrow \Sigma_{\downarrow featset}, \Sigma_{\downarrow y_i}$
6: $t \leftarrow$ CreateDecisionTree$(feat, lbl)$
7: **for** n $\in$ LeafNodes(t) **do**
8:   **if** Label(n) = 1 **then**
9:     $\pi \leftarrow$ Path$(t, root, n)$ {A path from root to node n in tree t}
10:     $f_i \leftarrow f_i \vee \pi$
11: **for** $y_k \in f_i$ **do**
12:   $d_k \leftarrow d_k \cup y_i \cup d_i$
13: **return** f_i, D

5) RepairHkF: This subroutine repairs the current candidate function vector to fix the counterexample. It takes the specification, candidate function vector, a counterexample, and Order, a linear extension of dependencies among Y variables as input, and returns a repaired candidate function vector. Algorithm 3 discusses RepairHkF subroutine in detail.

Algorithm 1 starts with generating samples Σ by calling GetSamples subroutine at line 1. Next, Algorithm 1 initializes the set D (line 2), which is a collection of d_i, where d_i represents the set of Y variables that depends on y_i. Lines 3-5 introduce variable ordering constraints based on the subset relations in each $\langle H_i, H_j \rangle$ pair, that is, if $H_j \subset H_i$, then y_i can depend on y_j. Line 7 calls the subroutine CandidateHkF for every y_i variable to learn the candidate function f_i. Next, at line 8, Manthan3 calls FindOrder to compute Order, a topological ordering among the Y variables that satisfy all the ordering constraints in D.

In line 11, CheckSat checks the satisfiability of the formula $E(X, Y')$ described at line 10. If $E(X, Y')$ is SAT, then Manthan3 at line 13 performs another satisfiability check to ensure that propositional model to X can be extended to Y. If CheckSat at line 13 is UNSAT, then Algorithm 1 terminates at line 15 as there does not exists a Henkin function vector, otherwise Manthan3 has a counterexample σ to fix. The candidate vector $\boldsymbol{f}$ goes into a repair iteration (line 17) based on the counterexample σ, that is, the subroutine RepairHkF repairs the current function vector $\boldsymbol{f}$ such that σ now gets fixed. Manthan3 returns a function vector $\boldsymbol{f}$ only if $E(X, Y')$ is UNSAT.

We now discuss the subroutines CandidateHkF and RepairHkF in detail.

Algorithm 2 shows the CandidateHkF subroutine. CandidateHkF assumes access to CreateDecisionTree that constructs a decision tree t from labeled data on a set of features $featset$. It uses the ID3 algorithm [25] and we used the Gini Index [25] as the impurity measure.

In Algorithm 2, line 1 includes the feature set, *featset*, for y_i in the dependency set H_i. Further, line 3 *extends* the features to include all the y_j variables that have the dependency set H_j as a subset of H_i, and y_j does not depend on y_i to allow the decision tree to learn over such y_j as well. Line 5 selects valuations of feature set and label from samples Σ, and learns a decision tree. Then, Lines 7-10 constructs a logical formula as a representation of the decision tree by constructing a disjunction over all paths in the tree that lead to class label **1**. In line 12, set d_k is updated for variable y_k that appears as a node in decision tree t for y_i.

Algorithm 3 RepairHkF$(\varphi(X,Y), \boldsymbol{f}, \sigma,$ Order$)$

1: $H \leftarrow \varphi(X,Y) \wedge (X \leftrightarrow \sigma[X])$; $S \leftarrow (Y \leftrightarrow \sigma[Y'])$
2: $Ind \leftarrow$ FindCandi(H, S)
3: **for** $y_k \in Ind$ **do**
4:   $\hat{Y} \leftarrow \emptyset$
5:   **for** $y_j \in Y$ **do**
6:     **if** $H_j \subseteq H_k \wedge$ Order$[index(y_j)] >$ Order$[index(y_k)]$ **then**
7:       $\hat{Y} \leftarrow \hat{Y} \cup y_j$
8:   $G_k \leftarrow (y_k \leftrightarrow \sigma[y'_k]) \wedge \varphi(X,Y) \wedge (H_k \leftrightarrow \sigma[H_k]) \wedge (\hat{Y} \leftrightarrow \sigma[\hat{Y}])$
9:   $ret, \rho \leftarrow$ CheckSat(G_k)
10:   **if** $ret = UNSAT$ **then**
11:     $C \leftarrow$ FindCore(G_k)
12:     $\beta \leftarrow \bigwedge_{l \in C} ite((\sigma[l] = 1), l, \neg l)$
13:     $f_k \leftarrow ite((\sigma[y'_k] = 1), f_k \wedge \neg\beta, f_k \vee \beta)$
14:   **else**
15:     **for** $y_t \in Y \setminus \hat{Y}$ **do**
16:       **if** $\rho[y_t] \neq \sigma[y'_t]$ **then**
17:         $Ind \leftarrow Ind.Append(y_t)$
18:     $\sigma[y_k] \leftarrow \sigma[y'_k]$
19: **return** $\boldsymbol{f}$

Algorithm 3 represents the RepairHkF subroutine. RepairHkF assumes access to the following subroutines:

1) FindCandi: It takes hard constraints and soft constraints as input. It makes a MaxSAT solver call on a specification containing hard and soft constraints and returns a set of variables corresponding to which the soft constraints are dropped by MaxSAT solver in order to satisfy the specification.
2) FindCore: It takes a UNSAT formula as an input and returns unsatisfiable core (UnsatCore) of the formula.

Algorithm 3 first attempts to find the potential candidates to repair using FindCandi. At line 2, FindCandi subroutine essentially calls a MaxSAT solver with $\varphi(X,Y) \wedge (X \leftrightarrow \sigma[X])$ as hard-constraints and $(Y \leftrightarrow \sigma[Y])$ as soft-constraints to find the potential candidates to repair, it returns a list (*Ind*) of Y variables such that candidates corresponding to each of the variables appearing in (*Ind*) are potential candidates to repair. For each of the $y_k \in$ *Ind*, line 6 computes $\hat{Y}$, which is a set of y_j variable that appears after y_k in Order and corresponding H_j is a subset of H_k.

Next, Algorithm 3 checks the satisfiability of the G_k formula at line 9. If G_k is UNSAT, line 11 attempts to find the UnsatCore of G_k using subroutine FindCore, and line 12 constructs a repair formula β, using the literals corresponding to unit clauses in UnsatCore. Depending on the value of $\sigma[y'_k]$, β is used to strengthen or weaken f_k at line 13. If G_k is SAT and $\rho \models G_k$, lines 15-18 look for other potential candidates to repair, and add all y_t variables for which $\rho[y_t]$ is not same as $\sigma[y'_t]$ to the list *Ind*.

Note that in line 8, we add a constraint $\hat{Y} \leftrightarrow \sigma[\hat{Y}]$ in $G_i(X,Y)$ where $\hat{Y}$ is a set of Y variables such that for all y_j of $\hat{Y}$, $H_j \subseteq H_i$. Fixing valuations for such y_j variables helps Manthan3 to synthesize a better repair for candidate f_i. Consider the following example. Let $\forall X \exists^{H_1} \exists^{H_2} \varphi(X,Y)$, where $\varphi(X,Y) : (y_1 \leftrightarrow x_1 \oplus y_2)$, $H_1 = \{x_1\}$ and $H_2 = \{x_1\}$. Let us assume that we need to repair the candidate f_1, and $G_1(X,Y) = (y_1 \leftrightarrow \sigma[y'_1]) \wedge \varphi(X,Y) \wedge (x_1 \leftrightarrow \sigma[x_1])$. As

$G_1(X, Y)$ does not include the current value of y_2 that led to the counterexample, it misses out on driving f_1 in a direction that would ensure $y_1 \leftrightarrow x_1 \oplus y_2$. In fact, in this case repair formula β would be empty, thereby failing to repair.

By definition of Henkin functions, we know that the following lemma holds:

Lemma 1 *$\boldsymbol{f}$ is a Henkin function vector if and only if $\neg\varphi(X, Y) \wedge (Y \leftrightarrow \boldsymbol{f})$ is UNSAT.*

Manthan3 returns a function vector only when $E(X, Y') : \neg\varphi(X, Y') \wedge (Y' \leftrightarrow \boldsymbol{f})$ is UNSAT, and each function f_i follows Henkin dependencies by construction. Therefore Manthan3 is sound, and returned function vector is a Henkin function vector.

Limitations: There are instances for which Manthan3 might not be able to repair a candidate vector, and consequently is not complete. The limitation is that the formula $G(X, Y)$ (Formula 1) is not aware of Henkin dependencies.

Let us consider an example, $\phi : \forall X \exists^{H_1} y_1 \ \exists^{H_2} y_2 \ \varphi(X, Y)$ where $X = \{x_1, x_2, x_3\}$, $Y = \{y_1, y_2\}$, $\varphi(X, Y) := \neg(y_1 \oplus y_2)$, $H_1 = \{x_1, x_2\}$, and $H_2 = \{x_2, x_3\}$. Note that ϕ is True and Henkin functions are $\boldsymbol{f} := \langle f_1(x_1, x_2) : x_2, f_2(x_2, x_3) : x_2 \rangle$. Let us assume the candidates learned by Manthan3 is $\boldsymbol{f} := \langle f_1(x_1, x_2) : x_2, \ f_2(x_2, x_3) : \neg x_2 \rangle$. The learned candidates are not Henkin functions as $E(X, Y')$ is SAT. Let the counterexample to repair is σ is $\langle x_1 \leftrightarrow 0, x_2 \leftrightarrow 0, x_3 \leftrightarrow 0, y_1 \leftrightarrow 0, y_2 \leftrightarrow 0, y_1' \leftrightarrow 0, y_2' \leftrightarrow 1 \rangle$.

Let the candidate to repair is y_2, and corresponding G_2 formula is $G_2 := \varphi(X, Y) \wedge (x_2 \leftrightarrow 0) \wedge (x_3 \leftrightarrow 0) \wedge (y_2 \leftrightarrow 1)$. As $H_1 \not\subseteq H_2$, the formula G_2 is not allowed to constrain on y_1. G_2 turns out SAT, suggesting that we should try to repair y_1 instead of y_2, but as y_1 is also not allowed to depend on y_2, the formula G_1 would also be SAT. Therefore, Manthan3 is unable to repair candidate $\boldsymbol{f}$ to fix counterexample the σ. Manthan3 would not be able to synthesize Henkin functions for such a case. Hence, Manthan3 is not complete.

V. Experimental Results

We implemented Manthan3[1] using Python, and it employs Open-WBO [23] for MaxSAT queries, PicoSAT [4] to find UNSAT cores, ABC [21] to represent and manipulate Boolean functions, CMSGen to generate the required samples [15], UNIQUE [27] to extract definition for uniquely defined variables, and Scikit-Learn [2] to learn the decision trees.

Instances: We performed an extensive comparison on 563 instances consisting of a union of instances from the DQBF track of QBFEval18, 19, and 20 [1], which encompass equivalence checking problems, controller synthesis, and succinct DQBF representations of propositional satisfiability problems.

Test hardware: 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 core with a timeout of 7200 seconds for each benchmark.

Tools compared with: We performed a comparison vis-a-vis the prior state-of-the-art techniques, HQS2 [11] and Pedant [26]. Note that we compared Manthan3 with the tools that can synthesize Henkin functions for True DQBF; the rest all the DQBF solvers, including DepQBF [22], DQBDD [28] do not synthesize such functions. The DQBF preprocessor HQSpre [31] is invoked implicitly by HQS2. We found that the performance of Pedant degrades with the preprocessor HQSPre; therefore, we consider the results of Pedant without preprocessing. Manthan3 is used without HQSpre.

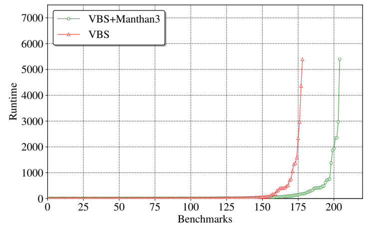

Fig. 1: Virtual Best Synthesizing Henkin functions with/without Manthan3. VBS in the plot represents VBS of HQS2 and Pedant. A point $\langle x, y \rangle$ implies that a tool took less than or equal to y seconds to synthesize a Henkin function vector for x many instances on a total of 563 instances.

Evaluation objective: It is well-known that different techniques are situated differently for different classes of instances in the context of NP-hard problems. The practical adoption often employs a portfolio approach [7], [18], [32]. Therefore, in practice, one is generally interested in evaluating the impact of a new technique on the portfolio of existing state-of-the-art tools. Hence, to evaluate the impact of our algorithm on the instances that the current algorithms cannot handle, we focus on the **Virtual Best Synthesizer** (VBS), which is the portfolio of the best of the currently known algorithms. If at least one tool in the portfolio could synthesize Henkin functions for a given instance, it is considered to be synthesized by VBS; that is, VBS is at least as powerful as each tool in the portfolio. The time taken to synthesize Henkin functions for the given instance by VBS is the minimum of the time taken by any tool to synthesize a function for that instance.

Results: Figure 1 represents the cactus plot for VBS of HQS2 and Pedant vis-a-vis with VBS of HQS2, Pedant, and Manthan3. We observe that the VBS with Manthan3 synthesizes functions for 204 instances while VBS without Manthan3 synthesizes functions for only 178 instances; that is, the VBS improves by 26 instances with Manthan3. Of 563 instances, for 204 instances, Henkin functions are synthesized by at least one of three tools. Manthan3 achieves the smallest synthesizing time on 42 instances, including 26 instances for which none of the other tools could synthesize Henkin functions. [2]

These 26 instances are mainly where Pedant and HQS2 struggle to scale. Considering the case of controller synthesis [5], Pedant and HQS2 struggle to synthesize Henkin functions as dependencies for winning state variables increase. Considering, *cnt11y* and *cnt30y* instances – cnt11y has 12 variables, whereas cnt30y has 31 variables as dependencies for winning state variable. Pedant, HQS2, and Manthan3 took 63.43, 144.16, and 3.28 seconds respectively to synthesize Henkin functions for

[1] Manthan3 is available at https://github.com/meelgroup/manthan

[2] Additional experiments, examples, and discussions are deferred to technical report [13].

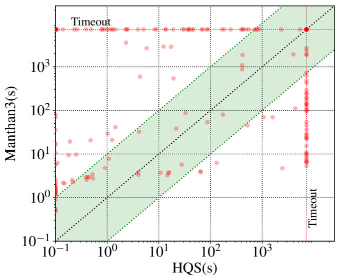

Fig. 2: Manthan3 vs. HQS2.

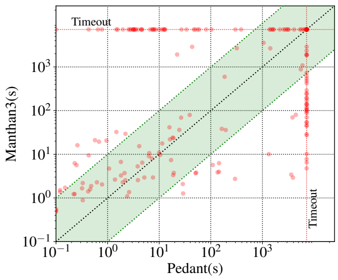

Fig. 3: Manthan3 vs. Pedant

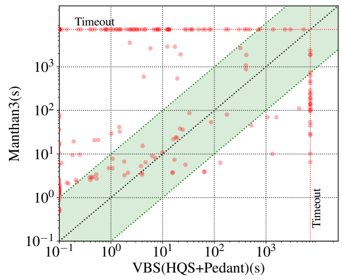

Fig. 4: VBS(HQS2+Pedant) vs. Manthan3.

A point $\langle x, y\rangle$ implies that the synthesizer on $\langle x\rangle$ axis took x sec. while the synthesizer on $\langle y\rangle$ axis tooks y sec. to synthesize Henkin functions for an instance.

cnt11y. However, for cnt30y, Pedant could not synthesize even with 27000 iterations, and HQS2 timed out while converting DQBF to QBF. Whereas, Manthan3 took only 12.22 seconds to synthesize.

Figure 4 highlights that the performance of Manthan3 is orthogonal to existing tools. Furthermore, as shown in green area of Figure 4, for 47 instances Manthan3 took less than or equal to additional 10 seconds to synthesize Henkin functions than by the VBS with HQS2 and Pedant.

Figure 2 (resp. Figure 3) represents scatter plot for Manthan3 vis-a-vis with HQS2 (resp. Pedant). The distribution of the instances for which functions are synthesized shows that all three tools are incomparable. There are many instances where only one of these tools succeeds, and others fail.

In total there are 148, 138 and 116 instances for which HQS2, Pedant and Manthan3 could synthesize Henkin functions respectively. Moreover, there are 40 instances for which Manthan3 could synthesize Henkin functions, whereas HQS2 could not. Similarly, there are 37 instances for which Pedant could not synthesize Henkin functions and Manthan3 synthesized. In total 88 instances for which Manthan3 was not able to synthesize functions, however, either Pedant or HQS2 could synthesize Henkin functions. Due to incompleteness of Manthan3, it could not handle 49 out of those 88 instances and for remaining instances it timed out.

Acknowledgements: This work was supported in part by National Research Foundation Singapore under its Campus for Research Excellence and Technological Enterprise (CREATE) programme, NRF Fellowship Programme [NRF-NRFFAI1-2019-0004], Ministry of Education Singapore Tier 2 grant [MOE-T2EP20121-0011], and Ministry of Education Singapore Tier 1 Grant [R-252-000-B59-114]. The computational work was performed on resources of the National Supercomputing Centre, Singapore https://www.nscc.sg.

References

[1] "QBF solver evaluation portal 2020," 2020. [Online]. Available: http://www.qbflib.org/qbfeval20.php
[2] "sklearn.tree.decisiontreeclassifier," 2021. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html
[3] V. Balabanov, H.-J. K. Chiang, and J.-H. R. Jiang, "Henkin quantifiers and boolean formulae: A certification perspective of dqbf," *Proc. of Theoretical Computer Science*, 2014.
[4] A. Biere, "PicoSAT essentials," *Proc. of JSAT*, 2008.
[5] R. Bloem, R. Könighofer, and M. Seidl, "Sat-based synthesis methods for safety specs," in *Proc. of VMCAI*, 2014.
[6] K. Chatterjee, T. A. Henzinger, J. Otop, and A. Pavlogiannis, "Distributed synthesis for ltl fragments," in *Proc. of FMCAD*, 2013.
[7] J. M. Dudek, V. H. N. Phan, and M. Y. Vardi, "ProCount: Weighted projected model counting with graded project-join trees," in *Proc. of SAT*, 2021.
[8] A. Fröhlich, G. Kovásznai, A. Biere, and H. Veith, "idq: Instantiation-based DQBF solving." in *Proc. of SAT*, 2014.
[9] M. R. Garey, "A guide to the theory of np-completeness," *Computers and intractability*, 1979.
[10] K. Gitina, S. Reimer, M. Sauer, R. Wimmer, C. Scholl, and B. Becker, "Equivalence checking of partial designs using dependency quantified boolean formulae," in *Proc. of ICCAD*, 2013.
[11] K. Gitina, R. Wimmer, S. Reimer, M. Sauer, C. Scholl, and B. Becker, "Solving dqbf through quantifier elimination," in *Proc. of DATE*, 2015.
[12] P. Golia, S. Roy, and K. S. Meel, "Manthan: A data-driven approach for Boolean function synthesis," in *Proc. of CAV*, 2020.
[13] ——, "Synthesis with explicit dependencies," 2023. [Online]. Available: https://arxiv.org/abs/2301.10556
[14] P. Golia, F. Slivovsky, S. Roy, and K. S. Meel, "Engineering an efficient boolean functional synthesis engine," in *Proc. of ICCAD*, 2021.
[15] P. Golia, M. Soos, S. Chakraborty, and K. S. Meel, "Designing samplers is easy: The boon of testers," in *Proc. of FMCAD*, 2021.
[16] R. Gupta, S. Sharma, S. Roy, and K. S. Meel, "WAPS: Weighted and projected sampling," in *Proc. of TACAS*, 2019.
[17] L. Henkin, "Some remarks on infinitely long formulas, infinitistic methods," 1959.
[18] H. H. Hoos, T. Peitl, F. Slivovsky, and S. Szeider, "Portfolio-based algorithm selection for circuit qbfs," in *Proc. of CP*, 2018.
[19] J.-H. R. Jiang, V. N. Kravets, and N.-Z. Lee, "Engineering change order for combinational and sequential design rectification," in *Proc. of DATE*, 2020.
[20] M. Krynicki and M. Mostowski, "Henkin quantifiers," in *Quantifiers: logics, models and computation*, 1995.
[21] B. Logic and V. Group, "ABC: A system for sequential synthesis and verification," 2021. [Online]. Available: http://www.eecs.berkeley.edu/~alanmi/abc/
[22] F. Lonsing and A. Biere, "DepQBF: A dependency-aware QBF solver," *Proc. of JSAT*, 2010.
[23] R. Martins, V. Manquinho, and I. Lynce, "Open-WBO: A modular MaxSAT solver," in *Proc. of SAT*, 2014.
[24] G. Peterson, J. Reif, and S. Azhar, "Lower bounds for multiplayer noncooperative games of incomplete information," *Computers & Mathematics with Applications*, 2001.
[25] J. R. Quinlan, "Induction of decision trees," *Proc. of Machine learning*, 1986.
[26] F.-X. Reichl, F. Slivovsky, and S. Szeider, "Certified DQBF solving by definition extraction," in *Proc. of SAT*, 2021.
[27] F. Slivovsky, "Interpolation-based semantic gate extraction and its applications to QBF preprocessing," in *Proc. of CAV*, 2020.
[28] J. Síč and J. Strejček, "DQBDD: an efficient BDD-based DQBF solver," in *Proc. of SAT*, 2021.
[29] L. Tentrup and M. N. Rabe, "Clausal abstraction for DQBF," in *Proc. of SAT*, 2019.
[30] K. Wimmer, R. Wimmer, C. Scholl, and B. Becker, "Skolem functions for QBF," in *Proc. of ATVA*, 2016.
[31] R. Wimmer, S. Reimer, P. Marin, and B. Becker, "Hqspre–an effective preprocessor for QBF and DQBF," in *Proc. of TACAS*, 2017.
[32] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Satzilla: portfolio-based algorithm selection for SAT," *Proc. of JAIR*, 2008.