Henkin Synthesis: DQBF meets Machine Learning

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 $\forall, \exists$ for propositional variables. In QBF an existentially quantified variable is allowed to depend on all universally quantified variables in its scope. Dependency Quantified Boolean Formulas (DQBF) restricts the dependencies of existentially quantified variables. In DQBF, existentially quantified variables have explicit dependencies on subset of universally quantified variables, called Henkin dependencies. Given a Boolean specification between the set of inputs and set of outputs, the problem of Henkin synthesis is to synthesise 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, the synthesis of safe controllers, analysis of games.**

In this work, we propose a machine learning-based approach to Henkin synthesis, called DepManthan. DepManthan is the first general-purpose technique that can synthesise Henkin functions for both True and False DQBF instances. On an extensive evaluation over 550 instances, we demonstrate that DepManthan can synthesise functions for 350 instances compared to 101 instances by the prior state-of-the-art.

I. Introduction

The satisfiability problem (SAT) checks for the existence of an assignment for the propositional variables of the formula φ such that, φ upon substituting the assignment evaluates to True. The NP revolution for SAT opened up further research in the problems that lie *beyond NP* from a complexity-theoretic perspective, including MaxSAT solving [22], Model counting [28], QBF solving [20], [21], [25]–[27], DQBF solving [11], [12], [31], [32].

In this work, we focus on Dependency Quantified Boolean Formulas (DQBF) representation, which equips the propositional logic with universal ($\forall$) and existential($\exists$) quantification over propositional variables wherein each of the existentially quantified variable is allowed to depend on a pre-defined set, also called Henkin dependencies, of universally quantified variables. Observe that the standard model of Quantified Boolean formulas (QBF) is a subset of QBF wherein an existentially quantified variable is allowed to depend on all the universally quantified variables. Our interest in DQBF stems from its diverse applications such as equivalence checking of partial functions [12], finding strategies for incomplete games [23], controller synthesis [8].

Recent years have seen a surging interest in DQBF solving [11], [12], [31], [32]. Solving DQBF is a decision problem that looks for an answer to the question, *Does for all the valuations of universally quantified variables, there exists a valuation for existentially quantified variables in accordance to their Henkin dependencies such that formula evaluates to True?* However, at times, just a True/False answer is not sufficient. For example, consider the scenario of circuit realizability, in addition to finding out whether a circuit is realizable or not, one would like to know the circuit implementation as well, which motivates the need for techniques to synthesize functions corresponding to existentially quantified variables.

Given a Boolean specification between a set of inputs and outputs, the problem of Henkin synthesis is to synthesise each output as a function of its Henkin dependencies such that the specification is met. Henkin synthesis is not restricted to only True DQBF instances; it needs to synthesise a function within the *satisfiable region* of Boolean specification. While there are diverse techniques for solving DQBF, we are not aware of any general purpose Henkin synthesiser that can synthesise Henkin functions for both True and False DQBF instances. Recently, Wimmer et al. [32] proposed a technique for obtaining Henkin functions for *only* True DQBF instances from elimination-based DQBF solvers.

In this work, we present the first general-purpose Henkin synthesiser, called DepManthan. DepManthan takes a machine learning based approach to generate the candidates, and it further uses the advances in automated reasoning to repair the candidates if needed. In an extensive comparison on 550 instances from diverse suites, DepManthan could synthesise Henkin functions for 350 instances, whereas the state-of-the-art Henkin function synthesiser (which synthesises Henkin functions only for True instances) could solve 101 instances.

As a Henkin function synthesiser can also be used as a DQBF solver with an additional SAT call, we further experimented with DepManthan as a DQBF solver. DQBF solving is not an ideal setting for DepManthan, as it looks for functions instead of *conflicts*. Surprisingly, DepManthan shows competitive performance with the state-of-the-art DQBF solvers. Significantly, DepManthan could solve 67 instances that are beyond the reach of the existing state of the art DQBF solver — adding 67 new instances to the portfolio of DQBF solvers.

II. Preliminaries

We used lower case letter to represents a propositional variable and a upper case letter to represents a set of variables. We used standard notation for logical gates as $\wedge, \vee, \neg$. A literal is either a variable or its negation, and a clause is considered as disjunction of literals. If a formula φ is a conjunction of clauses, it is considered in Conjunctive Normal Form (CNF).

Vars(φ) represents the set of variables appeared 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 used $\sigma \models \varphi$ 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.

The *unsatisfiable core* of a formula φ is a subset of clauses of φ for which there does not exists a satisfying assignment. We used UnsatCore to represents *unsatisfiable core*. 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 $\phi : \forall x_1 \ldots x_n \exists^{H_1} y_1 \ldots \exists^{H_m} y_m \varphi(X,Y)$ where $X = \{x_1,\ldots,x_n\}$, $Y = \{y_1,\ldots,y_m\}$ and H_i represents the dependency set of y_i, that is, variable y_i can only depend on H_i and $H_i \subset X$. $\exists^{H_i}$ is called *Henkin* quantifiers [15].

The satisfiable region of ϕ is a set of all assignments of universally quantified variables (X) for which there exists a valuation for existentially quantified variables (Y) such that $\varphi(X,Y)$ evaluates to True.

A DQBF formula ϕ is considered to be True if and only if for all assignments to dependencies H, there always exists a valuation of Y that satisfy $\varphi(X,Y)$, else it is considered to be False. That is, a DQBF is True, if there exits a function $f_i : \{0,1\}^{|H_i|} \mapsto \{0,1\}$ for each existentially quantified variable y_i, such that $\varphi(X, f_1(H_i), \ldots, f_m(H_m))$ which is obtained by replacing each y_i by its corresponding function f_i is tautology.

Henkin Synthesis: Given a DQBF formula,

$$\exists^{H_1} y_1. \ \ldots \exists^{H_m} y_m. \ \varphi(x_1,\ldots,x_n,y_1,\ldots,y_m)$$

where $x_1,\ldots,x_n \in X, y_1,\ldots,y_m \in Y$, $H_i \subseteq X$. Synthesise a function vector $\boldsymbol{f} : \langle f_1,\ldots,f_m\rangle$ such that

$$\exists^{H_1} y_1. \ \ldots \exists^{H_m} y_m. \ \varphi(X,Y) \equiv \varphi(X, f_1(H_1),\ldots,f_m(H_m))$$

$\boldsymbol{f}$ is called Henkin function vector and each f_i is called Henkin function.

There is mainly two key difference for solving DQBF instances and synthesising Henkin synthesis:

1) DQBF solving is a decision problem, it *just* checks with respect to the existence of a Henkin function vector, $\boldsymbol{f}$, such that $\varphi(X, f_1(H_1),\ldots,f_m(H_m))$ becomes tautology.
2) Henkin synthesis also includes synthesising functions for False DQBF instances. Consider the example $\phi := \forall X \exists y_1^{H_1} \varphi(x_1,x_2,y_1)$ where $\varphi(x_1,x_2,y_1) : \neg(x_1 \oplus x_2) \wedge (y_1 \leftrightarrow x_2)$, and $H_1 = \{x_1\}$. DQBF formula ϕ is False. However, for all valuation of X in the satisfiable region of $\phi : \exists^H Y \varphi(X,Y)$, i.e, $x_1 = 0, x_2 = 0$, and $x_1 = 1, x_2 = 1$, there exists a function $f_1(x_1) = x_1$ such that $\varphi(x_1,x_2,f_1(x_1))$ evaluates to True. That is,

$$\exists^{H_1 := \{x_1\}} y_1 \ \varphi(x_1,x_2,y_1) \equiv \varphi(x_1,x_2,f_1(x_1))$$

Therefore, $f_1(x_1)$ is a Henkin function for y_1.

A special case of DQBF is a 2-Quantified Boolean Formula (2-QBF), in which $H_1 = H_2 = \ldots = H_m = X$. For 2-QBF, function synthesis is called Skolem synthesis.

Skolem Synthesis: Given a 2-QBF formula, $\phi := \forall X \exists Y \varphi(X,Y)$ where $X = \{x_1,\ldots,x_n\}$ and $Y = \{y_1,\ldots,y_m\}$, Synthesise a function vector $\boldsymbol{f} : \langle f_1,\ldots,f_m\rangle$ such that

$$\exists Y \varphi(X,Y) \equiv \varphi(X, f_1(X),\ldots,f_m(X))$$

$\boldsymbol{f}$ is called Skolem function vector and each f_i is called Skolem function.

Observe that every Henkin function is also a Skolem function, whereas vice-a-versa is not true. Let us consider an example, let $\phi : \forall X \exists^{H_1} y_1 \varphi(X,Y)$ where $X = \{x_1,x_2\}, Y = \{y_1\}, H_1 = \{x_1\}$ and $\varphi(X,Y) := (x_1 \vee x_2 \vee y_1)$. Note that, for y_1, there exists multiple Skolem functions like $\neg(x_1 \vee x_2), \neg(x_1 \wedge x_2), \neg x_1, \neg x_2, 1$. However, now as we have $H_1 = \{x_1\}$, that is, y_1 is allowed to depend only on x_1, the possible Henkin functions for y_1 is $\neg x_1$ and 1. The Henkin dependencies restricts the space of possible functions.

Furthermore, added Henkin dependencies can reduce the satisfiable region for ϕ. For better understanding, let us consider an another example, let $\phi : \forall x_1,x_2 \exists y_1 \varphi(x_1,x_2,y_1)$, where $\varphi(x_1,x_2,y_1) : (y_1 \leftrightarrow (x_1 \vee x_2))$. ϕ is a 2-QBF formula, and for all valuations of x_1 and x_2, there exists a valuation of y_1, therefore ϕ is True. Now, let us consider the case where we have restricted dependencies for y_1, $\phi : \forall x_1,x_2 \exists^{H_1} y_1 \ (y_1 \leftrightarrow (x_1 \vee x_2))$, where $H_1 = x_1$. ϕ is a DQBF formula, and y_1 can only depend on x_1, that is, in some sense it can only *see* the valuation of x_1. There does not exists a function, $f_1(x_1)$ such that $(f_1(x_1) \leftrightarrow (x_1 \vee x_2))$ turns out to be a tautology, therefore ϕ is False.

From now onwards, we say that X is an input variable set, Y is a output variable set, and $\varphi(X,Y)$ represents Boolean relation between inputs and outputs. We used $\exists^{H_1} y_1 \ldots \exists^{H_m} y_m \varphi(x_1,\ldots,x_n,y_1,\ldots,y_m)$ and $\exists^H \varphi(X,Y)$ interchangeably.

The error formula (Formula 1) purposed by John et al. [18] can be used to verify whether $\boldsymbol{f}$ is Henkin function or not.

$$E(X,Y,Y') = \varphi(X,Y) \wedge \neg\varphi(X,Y') \wedge (Y' \leftrightarrow \boldsymbol{f})$$
$$\text{Where } \boldsymbol{f} \text{ is function vector } \langle f_1,\ldots,f_{|Y|}\rangle \quad (1)$$

Lemma 1: If $E(X,Y,Y')$ is UNSAT and $\boldsymbol{f}$ follows the Henkin dependencies, then $\boldsymbol{f}$ is a Henkin function vector.

However, if $\boldsymbol{f}$ is Henkin function vector, error formula would not necessarily be UNSAT, in contrast to Skolem functions for which a $\boldsymbol{f}$ is Skolem function vector **if and only if** error formula is UNSAT [18]. The error-formula looks for an assignment of inputs X in the satisfiable region of $\exists Y \varphi(X,Y)$ to check whether for that valuation of X, $\varphi(X,\boldsymbol{f})$ evaluates to True or not. If the satisfiable regions for $\exists^H Y \varphi(X,Y)$ and $\exists Y \varphi(X,Y)$ are not same, error formula might not becomes UNSAT.

Corollary 1: If $\boldsymbol{f}$ is a Henkin function vector, and satisfiable regions of $\exists^H Y \varphi(X,Y)$ and $\exists Y \varphi(X,Y)$ are same, then error formula $E(X,Y,Y')$ is UNSAT.

III. Background

This work builds on top of the state-of-the-art data-driven technique for Skolem functional synthesis, called Manthan [14]. Manthan uses the advances of constraint-sampling, machine-learning and automated reasoning for efficient synthesis of Skolem functions. We now present a quick overview of Manthan.

Given a 2-QBF instance $\exists Y\, \varphi(X, Y)$, Manthan [14] synthesizes Skolem functions corresponding to output variables Y as follows:

Data Generation As the first step, Manthan samples the satisfying assignments of φ using an *a*daptative weighted sampling strategy. The generated samples would be considered as data to learn candidates in the later stages of Manthan.

Candidate Learning The generated samples are used to learn an approximate candidate vector,$\boldsymbol{f}$. The function, f_i, corresponding to each output variable y_i, is learned as a decision tree classifier. The valuations of X and Y in samples generated (allowed by certain variable ordering constraints, described next) are considered as the features, while the respective valuation of y_i is marked as the label.

Variables y_i and y_j satisfy an ordering constraint $y_i \prec_d y_j$; if y_j appears as the decision node in the decision tree learned for the candidate f_i corresponding to y_i. Manthan discovers requisite variable ordering constraints (among Y variables) on the fly as the candidate functions are learned. Given the partial variable ordering, Manthan extracts a *TotalOrder* for a valid variable ordering among Y variables.

Verification and Repair Manthan then verifies if the synthesised candidate vector $\boldsymbol{f}$ is a Skolem function vector by a satisfiability check on the *error formula* $E(X, Y, Y')$(1). If $E(X, Y, Y')$ is UNSAT, the candidates are Skolem functions and Manthan returns function vector $\boldsymbol{f}$. Otherwise, the candidates are *repaired* and the verification check is repeated.

If the candidate vector is not verified, Manthan uses the generated counterexample σ, that is, $\sigma \models E(X, Y, Y')$, to *repair* the candidate vector. To repair the candidate function f_i corresponding to output variable y_i, Manthan checks the satisfiability of $G_i(X, Y)$ (2):

$$\begin{gathered} G_i(X,Y) := \varphi(X,Y) \wedge (X \leftrightarrow \sigma[X]) \wedge (\hat{Y} \leftrightarrow \sigma[\hat{Y}]) \\ \wedge\, (y_i \leftrightarrow \sigma[y_i']) \\ \text{where } \sigma \models E(X,Y,Y'), \hat{Y} \subset Y, \text{ and} \\ \hat{Y} = \{\mathit{TotalOrder}[\mathit{index}(y_i)+1], \cdots, \mathit{TotalOrder}[|Y|]\} \end{gathered} \tag{2}$$

If $G_i(X, Y)$ is UNSAT, Manthan looks for unit clauses corresponding to X and $\hat{Y}$ in UnsatCore to construct a *repair formula* (β). If $G_i(X, Y)$ (2) is SAT, Manthan looks for other candidates to repair. Depending on the current valuation of the candidate function, Manthan would strengthen or weaken the candidate using the repair formula β.

IV. Related Work

From a theoretical perspective, the satisfiability problem of DQBF is known to be NEXPTIME-complete [23], in comparison to QBF which is *only* PSPACE-complete []. Recent year has seen significant interest in DQBF solving [10], [13], [31]. The first DPLL based approach to solve DQBF was proposed by Frohlich et al. [10]. Following the similar direction, recently Tentrup and Rabe [31] introduced the idea of using clausal abstraction for DQBF solving. Gitina et al. [12] proposed the idea of solving DQBF instance using a basic variable elimination strategy which is about transforming DQBF instance to a QBF by eliminating a set of variables that causes non-linear dependencies. The strategy is further improved by several optimization in [13]. Similar type of approach was proposed by Frohlich et al. [11] in which DQBF instance is transformed into a SAT instance by doing a local universal expansion on each clause.

There has been significant improvement in DQBF solving in recent years, however, extracting Henkin functions from DQBF instances remains the holy grail. We are not aware of any work that considers the computation of Henkin functions for both True and False instances. Wimmer at el. [32] proposed a method for obtaining Henkin functions for True instances from DQBF solvers that are based on variable elimination-based DQBF solving techniques [11], [13]. Elimination-based DQBF solvers execute a sequence of transformation of eliminating quantifiers on a DQBF instance to obtain a SAT instances, in this process, they obtain a sequence of equisatisfiable formulas $\varphi_1, \varphi_2, \ldots, \varphi_k$, where φ_i formula is a result of a transformation of φ_{i-1}. Wimmer at el. [32] showed that for a Ture DQBF instances, Henkin function for φ_{i-1} can be obtained from φ_i.

On the other hand, the area of extracting Skolem functions from QBF instances is well studied in the literature. The algorithmic progress for Skolem functional synthesis has been driven by a diverse set of techniques, including Knowledge representation based techniques [4], [5], [9], [18], [30], using incremental determinization in QBF solver [25], [26], extracting Skolem function from proof of validity of True QBF [6], [16], [17], and data-driven synthesis of Skolem functions [14].

V. Overview

DepManthan builds on top of Manthan [14] to handle additional Henkin dependencies constraints. The Henkin dependencies restricts the space of possible functions. We need to take care of these additional dependencies while learning and repairing the candidates to synthesise Henkin functions.

Figure 1 represents overview of DepManthan. As shown in Figure 1, DepManthan first generate data, which is the satisfying assignments of φ, and uses that data to learn a candidate vector $\boldsymbol{f}$, while taking care of Henkin dependencies. DepManthan verifies the candidate vector $\boldsymbol{f}$. If candidate vector passes the verification checks, DepManthan returns Henkin function vector, else candidate vector needs to undergo a repair iterations. We now discuss in detail about different components of DepManthan.

Candidate Learning: With the generated data, DepManthan learns candidates via a decision tree classifier. To learn a candidate f_i, DepManthan considers the valuation of f_i in

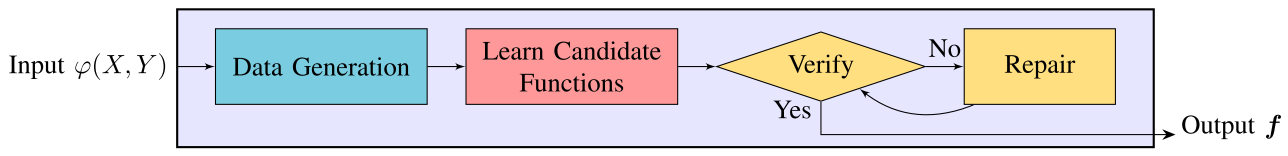

Fig. 1: Overview of DepManthan

data as label for the decision tree classifier. The feature set for f_i must be restricted only to H_i. However, we can include all the y_j to feature for which $H_j \subseteq H_i$. The function f_j is 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 cyclic dependencies, that is, if y_j appears in the learned candidate f_i, then y_i is not allowed as features to learn candidate f_j. If y_j appear in f_i, then we say y_i depends on y_j, denoted as $y_i \prec_d y_j$.

Variable Ordering: As mentioned in section III, if $\boldsymbol{f}$ is a valid function vector, then there exists a partial order $\prec_d$ over $\{y_1, \ldots, y_{|Y|}\}$ such that $y_i \prec_d y_j$. In the case of DepManthan, y_i can depend on y_j only if $H_j \subseteq H_i$. DepManthan extracts a valid *TotalOrder* from the partial dependencies learned in *candidate learning* among Y variables.

Verification: As a verification check, DepManthan checks the satisfiability of error formula(formula 1). If error formula is UNSAT, DepManthan outputs function vector $\boldsymbol{f}$. Else, when $E(X,Y,Y')$ is SAT, the satisfying assignment of $E(X,Y,Y')$ is used to identify and repair the candidate f_i's. Let us assume $\sigma \models E(X,Y,Y')$, we consider σ as a counterexample to fix.

Candidate Repair: To find which candidate to repair, DepManthan makes MaxSAT query with $\varphi(X,Y) \land (X \leftrightarrow \sigma[X])$ as hard constraints and $(Y \leftrightarrow \sigma[Y'])$ as soft constraints. Candidates for Y variable for which corresponding soft constraints are unsatisfiable undergo repair.

While repairing a candidate f_i corresponding to y_i with respect to σ, DepManthan constructs an another formula, called $G_i(X,Y)$. $G_i(X,Y)$ attempts to find the *reason* behind $\varphi(X,Y)$ being UNSAT with counterexample valuation of X ($\sigma[X]$) and Y' ($\sigma[Y']$). Note that Y' represents the outputs of $\boldsymbol{f}$ with $\sigma[X]$. The UnsatCore of $G_i(X,Y)$ represents the *reason*, and could be used to repair the candidate f_i. To ensure the Henkin dependencies, $G_i(X,Y)$ should be constrained over H_i instead of entire input set X. In addition to H_i, we can constrain G_i with y_j variables for which $H_j \subset H_i$. Also, $G_i(X,Y)$ can be constrained over all y_j with $H_j = H_i$, and *TotalOrder*$([index(y_j)]) >$ *TotalOrder*$([index(y_i)])$.

$$\begin{aligned} G_i(X,Y) := \varphi(X,Y) \land (H_i \leftrightarrow \sigma[H_i]) \land (\hat{Y} \leftrightarrow \sigma[\hat{Y}]) \\ \land (y_i \leftrightarrow \sigma[y_i']) \\ \text{where } \sigma \models E(X,Y,Y'), \forall y_j \in \hat{Y} : H_j \subseteq H_i \\ \text{and } \{\textit{TotalOrder}[index(y_j)] > \textit{TotalOrder}[index(y_i)]\} \end{aligned} \quad (3)$$

Variables corresponding to unit clauses in UnsatCore of $G_i(X,Y)$ is used to construct a repair formula, β. Depending on the valuation of y_i in counterexample, β is used to strengthen or weaken the candidate f_i.

Constraining $G_i(X,Y)$ over all y_j, whenever $H_j \subseteq H_i$, is necessary to synthesise a repair. Consider the $\exists^{H_1:=\{x_1\}}\exists^{H_2:=\{x_1\}}\varphi(X,Y)$, where $\varphi(X,Y) := (y_1 \leftrightarrow x_1 \oplus y_2)$. Let us assume, we need to repair candidate f_1, and we do not constrain over y_2 to construct $G_1(X,Y) = (y_1 \leftrightarrow \sigma[y_1']) \land \varphi(X,Y) \land (x_1 \leftrightarrow \sigma[x_1])$. 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 our repair formula β would be empty, thereby failing to synthesise a repair.

It may happen that $G_i(X,Y)$ formula, turns-out to be SAT, in that case, DepManthan attempts to find another candidates to repair. Let us assume $\rho \models G_i(X,Y)$, all y_j variables for which $\rho[y_j]$ not same as $\sigma[y_j']$ are considered as candidates to repair.

VI. Algorithmic Description

We now provide a detailed technical description of the pseudo-code of DepManthan (Algorithm 1).

Algorithm 1: DepManthan($\exists^{H_1} y_1 \ldots \exists^{H_m} y_m.\varphi(X,Y)$)

```
1  $\Sigma \leftarrow$ GetSamples($\varphi(X,Y)$)
2  $D \leftarrow \{d_1 = \emptyset \ldots, d_{|Y|} = \emptyset\}$
3  foreach $\langle H_i, H_j \rangle$ do
4      if $H_j \subset H_i$ then
5          $d_j \leftarrow d_j \cup y_i$
6  foreach $y_i \in Y$ do
7      $f_i, D \leftarrow$CandidateSkF($\Sigma, \varphi(X,Y), y_i, D$)
8  $TotalOrder \leftarrow$ FindOrder($D$)
9  repeat
10     $E(X,Y,Y') \leftarrow \varphi(X,Y) \land \neg\varphi(X,Y') \land (Y' \leftrightarrow \boldsymbol{f})$
11     $ret, \sigma \leftarrow$ CheckSat($E(X,Y,Y')$)
12     if $ret$ = $SAT$ then
13         $\boldsymbol{f} \leftarrow$ RepairSkF($\varphi(X,Y), \boldsymbol{f}, \sigma, TotalOrder$)
14 until $ret$ = $UNSAT$
15 $\boldsymbol{f} \leftarrow$ Substitute($F(X,Y), \boldsymbol{f}, TotalOrder$)
16 return $\boldsymbol{f}$
```

Algorithm 1 takes a DQBF instances $\exists^{H_1} y_1 \ldots \exists^{H_m} y_m \varphi(X,Y)$ as input and outputs a Henkin function vector $\boldsymbol{f} := \langle f_1, \ldots, f_m \rangle$. Algorithm 1 starts with generating samples from $\varphi(X,Y)$. At line 1, it calls subroutine GetSamples, which takes $\varphi(X,Y)$ as input to return the required samples Σ.

At line 2, set of sets D is initialized. D 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.

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

```
1  $featset \leftarrow H_i$
2  foreach $y_j \in Y$ 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  foreach $n \in$ LeafNodes($t$) do
8      if Label($n$) = 1 then
9          $\pi \leftarrow$ Path($t, root, n$)
10         $f_i \leftarrow f_i \vee \pi$
11 foreach $y_k \in f_i$ do
12     $d_i \leftarrow d_i \cup y_k \cup d_k$
13 return $f_i, D$
```

Algorithm 3: RepairSkF($\varphi(X,Y), \boldsymbol{f}, \sigma, TotalOrder$)

```
1  $H \leftarrow \varphi(X,Y) \wedge (X \leftrightarrow \sigma[X])$; $S \leftarrow (Y \leftrightarrow \sigma[Y'])$
2  $Ind \leftarrow$ FindRepairCandidates($H, S$)
3  foreach $y_k \in Ind$ do
4      $\hat{Y} \leftarrow \{TotalOrder[(y_k)+1], \cdots, TotalOrder[|Y|]\}$
5      $\hat{Y}' \leftarrow \emptyset$
6      foreach $y_j \in Y$ do
7          if $(H_j \subseteq H_k) \wedge (y_j \in \hat{Y})$ then
8              $\hat{Y}' \leftarrow \hat{Y}' \cup y_j$
9      $G_k \leftarrow (y_k \leftrightarrow \sigma[y'_k]) \wedge \varphi(X,Y) \wedge (H_k \leftrightarrow \sigma[H_k]) \wedge (\hat{Y}' \leftrightarrow \sigma[\hat{Y}'])$

10     $ret, \rho \leftarrow$ CheckSat($G_k$)
11     if $ret = UNSAT$ then
12         $C \leftarrow$ FindCore($G_k$)
13         $\beta \leftarrow \bigwedge_{l \in C} ite((\sigma[l]=1), l, \neg l)$
14         $f_k \leftarrow ite((\sigma[y'_k]=1), f_k \wedge \neg\beta, f_k \vee \beta)$
15     else
16         foreach $y_t \in Y \setminus \hat{Y}$ do
17             if $\rho[y_t] \neq \sigma[y'_t]$ then
18                 $Ind \leftarrow Ind.Append(y_t)$
19         $\sigma[y_k] \leftarrow \sigma[y'_k]$
20 return $\boldsymbol{f}$
```

Line 7 calls the subroutine CandidateSkF for every y_i variable to learn the candidate function f_i. Next, at line 8, DepManthan calls FindOrder to compute *TotalOrder*, a topological ordering among the Y variables that satisfy all the ordering constraints in D. In line 11, CheckSat checks the satisfiability of the *error formula* $E(X,Y,Y')$ described at line 10. If $E(X,Y,Y')$ is UNSAT, then DepManthan returns the functions, else the candidate function vector goes into a repair iteration based on the counterexample σ, at line 13, subroutine RepairSkF is called to repair function vector $\boldsymbol{f}$.

We now discuss the subroutines CandidateSkF and RepairSkF in details.

Algorithm 2 represents CandidateSkF subroutine. CandidateSkF assume access to following subroutines:

- CreateDecisionTree takes a features and label data, and outputs a decision tree t. It uses ID3 algorithm [24] to learn decision trees, and we used Gini Index [24] as impurity measure.
- Label takes a node n of decision tree and return class label of n.
- Path takes a decision tree t, root node of t and a node n, and return the path from root node to node n in decision tree t.

In Algorithm 2, line 1 sets the feature set, *featset*, for y_i to H_i, and 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 allows the decision tree to learn over such y_j as well. Lines 7-10, iterates over all leaf node n of class label 1, and disjunct the candidate f_i with path π, which is a path from root of decision tree t to leaf node n. Finally, at line 12, set d_i for y_i is updated with the Y variables which appeared as node in decision tree t.

Algorithm 3 represents the RepairSkF subroutine. Algorithm 3 first attempts to find the potential candidates to repair using subroutine FindRepairCandidates.At line 2, the FindRepairCandidates 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, FindRepairCandidates 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 4 computes $\hat{Y}$, a set of Y variables that appears after y_k in *TotalOrder*. Line 7 attempts to find $\hat{Y}'$, a subset of $\hat{Y}$ such that for every y_j in $\hat{Y}'$ corresponding H_j is a subset of H_k. Line 9 constructs G_k by constraining the repair candidate f_k over H_k and $\hat{Y}'$.

Next, Algorithm 3 checks the satisfiability of G_k formula at line 10. If G_k formula is UNSAT, line 12 attempts to find the UnsatCore of G_k using subroutine FindCore, and line 13 constructs a repair formula β, using the literals corresponding to unit clauses in UnsatCore. Depending on value of $\sigma[y'_k]$, β formula is used to strengthen or weaken the f_k at line 14. If G_k is SAT and $\rho \models G_k$, lines 16-19 looks 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 list*I nd*.

It is worth emphasizing that DepManthan builds on top of Manthan and continues to employ the low-level crucial data structures of Manthan; therefore, we refer the reader to [14] for such details. (The subroutine FindRepairCandidates is referred to as MaxSATList in [14]).

A. Example

We now illustrate Algorithm 1 through an example.

Example 1: Let $X = \{x_1, x_2, x_3\}$, $Y = \{y_1, y_2, y_3\}$ in $\exists^{H_1} y_1 \exists^{H_2} y_2 \exists^{H_3} y_3 \varphi(X,Y)$ where $\varphi(X,Y)$ is $(x_1 \vee y_1) \wedge (y_2 \leftrightarrow (y_1 \vee \neg x_2)) \wedge (y_3 \leftrightarrow (x_2 \vee x_3))$, and $H_1 = \{x_1\}, H_2 = \{x_1, x_2\}$, and $H_3 = \{x_2, x_3\}$.

1) DepManthan generates training data through sampling (Figure 2). As $H_1 \subset H_2$, DepManthan adds dependency that y_1 can not depend on y_2. DepManthan now attempts to learn candidates and call CandidateSkF for each y_i. As y_1 can not depend on y_2 and y_3 due to Henkin dependencies, the feature set for y_1 only includes H_1. The decision tree construction uses the samples of $\{x_1\}$ as features and samples of $\{y_1\}$ as labels. The candidate function f_1 is constructed by taking a disjunction over

x_1	x_2	x_3	y_1	y_2	y_3
0	0	0	1	1	0
0	0	1	1	1	1
1	1	0	0	0	1

Fig. 2: Samples of $\varphi(X, Y)$

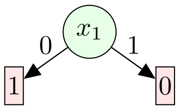

Fig. 3: Decision tree for y_1

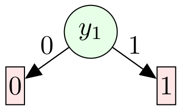

Fig. 4: Decision tree for y_2

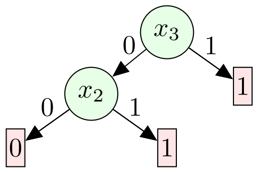

Fig. 5: Decision tree for y_3

all paths that end in leaf nodes with label 1: as shown in Figure 3, f_1 is synthesised as $\neg x_1$

As $H_1 \subset H_2$, the feature set for y_2 is H_2, y_1. The decision tree construction uses the samples of $\{x_1, x_2, y_1\}$ as features and samples of $\{y_2\}$ as labels. The candidate function f_1 is constructed by taking a disjunction over all paths that end in leaf nodes with label 1: as shown in Figure 4, f_2 is synthesised as y_1. Similarity, for y_3, the feature set is H_3, and a decision tree is constructed as shown in Figure 5 with samples of x_2, x_3 as features and samples of y_3 as label. We get synthesised function $f_3 := x_3 \lor (\neg x_3 \land x_2)$.

At the end of CandidateSkF, we have $f_1 := \neg x_1,\ f_2 := y_1,\ f_3 := x_3 \lor (\neg x_3 \land x_2)$. Let us assume the total order returned by FindOrder is $TotalOrder = \{y_3, y_2, y_1\}$.

2) We construct the error formula, $E(X, Y, Y') = \varphi(X, Y) \land \neg\varphi(X, Y') \land (Y' \leftrightarrow \Psi)$, which turns out to be SAT with counterexample $\sigma = \langle x_1 \leftrightarrow 1,\ x_2 \leftrightarrow 0,\ x_3 \leftrightarrow 0,\ y_1 \leftrightarrow 0,\ y_2 \leftrightarrow 1,\ y_3 \leftrightarrow 0,\ y'_1 \leftrightarrow 0,\ y'_2 \leftrightarrow 0,\ y'_3 \leftrightarrow 0\rangle$.

FindRepairCandidates calls a MaxSAT solver with $\varphi(X, Y) \land (x_1 \leftrightarrow \sigma[x_1]) \land (x_2 \leftrightarrow \sigma[x_2]) land(x_3 \leftrightarrow \sigma[x_3])$ as hard constraints and $(y_1 \leftrightarrow \sigma[y'_1]) \land (y_2 \leftrightarrow \sigma[y'_2]) \land (y_3 \leftrightarrow \sigma[y'_3])$ as soft constraints. FindRepairCandidates returns $ind = \{y_2\}$. Repair synthesis commences for f_2 with a satisfiability check of $G_2 = \varphi(X, Y) \land (x_1 \leftrightarrow \sigma[x_1]) \land (x_2 \leftrightarrow \sigma[x_2]) \land (y_1 \leftrightarrow \sigma[y'_1]) \land (y_2 \leftrightarrow \sigma[y'_2])$. Notice, here we can constrain G_2 with y_1 as $H_1 \subset H_2$. The formula is unsatisfiable, and DepManthan calls FindCore, which returns variable $\neg x_2$, since the constraints $(x_2 \leftrightarrow \sigma[x'_2])$ and $(y_2 \leftrightarrow \sigma[y'_2])$ are not jointly satisfiable in G_2. As the output f_2 for the assignment σ must change from 0 to 1, f_2 is repaired by disjoining with $\neg x_1$, and we get $f_2 := y_1 \lor \neg x_2$ as the new candidate. For the updated candidate vector $\boldsymbol{f}$ the error formula is UNSAT, and thus $\boldsymbol{f}$ is returned as a Henkin function vector.

B. DepManthan *as DQBF solver*

A Henkin synthesiser can easily be transformed into a DQBF solver. If $\phi : \forall X \exists^H Y \varphi(X, Y)$ is True, then $\varphi(X, \boldsymbol{f})$ should be a tautology, where $\varphi(X, \boldsymbol{f})$ represents the formula φ after substituting y_i with its Henkin function f_i. Once, DepManthan has synthesised Henkin function vector $\boldsymbol{f}$, it can check satisfiability of $\neg\varphi(X, \boldsymbol{f})$, that is, it calls CheckSat subroutine with $\neg\varphi(X, \boldsymbol{f})$ formula. If CheckSat returns UNSAT, then $\varphi(X, \boldsymbol{f})$ is a tautology and $\forall X \exists^H Y \varphi(X, Y)$ is True; else, it is False.

C. Limitations of DepManthan

There are mainly two limitations for DepManthan:

Inadequacy of Error Formula: Due to inadequacy of error formula (Formula 1), DepManthan fails to return Henkin functions for some cases. From Corollary 1, DepManthan can not return a Henkin function vector if the satisfiable regions for $\exists^H Y \varphi(X, Y)$ and $\exists Y \varphi(X, Y)$ are not same.

Failing to Repair Candidates: For some cases DepManthan might not be able to repair a candidate vector. Let us consider an example, $\phi : \forall X \exists^{H_1 := \{x_1, x_2\}} y_1 \exists^{H_2 := \{x_2, x_3\}} y_2 \varphi(X, Y)$ where $X = \{x_1, x_2, x_3\}$ $Y = \{y_1, y_2\}$ and $\varphi(X, Y) := \neg(y_1 \oplus y_2)$. Note that ϕ is True and a Henkin vector $\boldsymbol{f} : \langle f_1(x_1, x_2) = x_2, f_2(x_2, x_3) = x_2\rangle$.

Let us assume the candidate vector learned by DepManthan is $\boldsymbol{f} := \langle f_1(x_1, x_2) = x_2,\ f_2(x_2, x_3) = \neg x_2\rangle$. As we can see, error formula $E(X, Y, Y')$ is SAT. Let $\sigma \models E(X, Y, Y')$, and $\sigma = \langle x_1 \leftrightarrow 0,\ x_2 \leftrightarrow 0,\ x_3 \leftrightarrow 0,\ y_1 \leftrightarrow 1,\ y_2 \leftrightarrow 1,\ y'_1 \leftrightarrow 0,\ y'_2 \leftrightarrow 1\rangle$. Let the candidate to repair y_2. G_2 formula is constructed as follows:

$$G_2 := \varphi(X, Y) \land (x_1 \leftrightarrow 0) \land (x_2 \leftrightarrow 0) \land (x_3 \leftrightarrow 0) \land (y_2 \leftrightarrow 1)$$

As $H_1 \not\subset H_2$, G_2 formula is not allowed to constrain on y_1. G_2 formula turns-out to be 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, G_1 formula would also be SAT. Therefore, DepManthan is unable to repair candidate vector $\boldsymbol{f}$ to fix counterexample σ.

However, DepManthan returns Henkin function vector only if error formula (Formula 1) is UNSAT. Furthermore, $\boldsymbol{f}$ follows the Henkin dependencies by construction. Therefore, DepManthan is sound.

Theorem 1: DepManthan is sound but not complete.

VII. Experimental Results

We evaluate the performance of DepManthan on the union of benchmarks from DQBF track of QBFEval-19 [1] and QBFEval-20 [2]. We compare DepManthan with the state-of-the-art DQBF solvers DQBDD [29], DCAQE [31], HQS2 [13][1]. DCAQE was used with HQSPre [33] as a preprocessor, which is also implicitly used by DQBDD and HQS2. We used Open-WB0 [22] for MaxSAT queries, PicoSAT [7] to find UNSAT cores, ABC [19] to represent and manipulate Boolean functions, and Scikit-Learn [3] to learn decision trees.

All our experiments were conducted on a high-performance computer cluster with each node consisting of a E5-2690 v3

[1]We used the latest version of DCAQE,HQS2,DQBDD. DQBDD version 1.2 released on 10 April,2021.

CPU with 24 cores and 96GB of RAM, with a memory limit set to 4GB per core. All tools were run in a single-threaded mode on a single core with a timeout of 7200 seconds.

We compared DepManthan in two settings:

- **As a Henkin functional synthesizer:** As discussed in Section IV, a certificate producing DQBF solver can produce Henkin functions, but only for the True instances. For True instances, in comparison to certificate generating DQBF solver, HQS2 which could synthesise Henkin function for 101 instances, DepManthan was able to synthesise functions for 168 instances. Additionally, DepManthan could produced Henkin functions for 182 False instances.
- **As a DQBF solver:** Though not the ideal algorithm, a Henkin function synthesiser can also be used as a DQBF solver. We did an experiment with DepManthan as a DQBF solver. Note that DepManthan does not search conflicts but it first attempts to synthesise a Henkin function vector and then make a final check to return True or False — not an ideal setting for a DQBF solver. Despite this disadvantage, it is surprising that DepManthan shows a competitive performance with respect to diverse state of the art DQBF solving techniques.

Furthermore, we analyzed the impact of DepManthan in a setting of an ideal portfolio of state-of-the-art DQBF solvers. We observe that DepManthan solves **67** instances that could not be solved by any state-of-the-art tools— adding **67** new instances to a virtual best solver.

We now discuss the results in detail.

A. *Comparison with Henkin Functional Synthesis Engines*

We now present the comparison of DepManthan with the state-of-the-art Henkin function synthesiser, HQS2 [13] (HQS2 cannot produce Henkin functions for False instances). Figure 6 shows a cactus plot to compare the performance of DepManthan and HQS2. In figure 6, the number of instances are shown on the x-axis and the time taken on the y-axis; a point (x,y) in Figure 6 implies that a solver took less than or equal to y seconds to synthesise a Henkin function for x many instances on a total of 550 instances.

As shown in Figure 6, DepManthan significantly improves the state of the art for Henkin synthesis as DepManthan could synthesise Henkin functions for 350 instances out of 550, while HQS2 could synthesise for 101 instances — an improvement of 249 instances.

B. *Comparison with DQBF Solvers*

We now present the comparison of DepManthan as DQBF solver with the state-of-the-art solvers like DQBDD [29], DCAQE [31], HQS2 [13]. We used DCAQE with DQBF preprocessor HQSPre [33], while DQBDD and HQS2 use HQSPre implicitly.

Figure 7 represents the cactus plot for different DQBF solvers. As shown in Figure 7, DepManthan could solve 350 instances which is *only* 11 less than that of state of the art DQBF solver, DQBDD. It is surprising to see the competitive

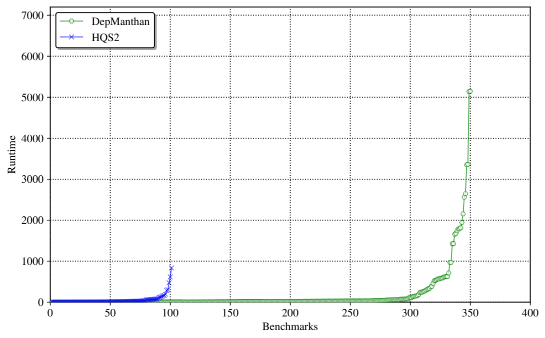

Fig. 6: DepManthan vs other Henkin function synthesiser.

TABLE I: DepManthan vs other state-of-the-art DQBF solvers.

	HQS2	DCAQE	DQBDD	DepManthan
Total(/550)	283	325	361	350
True	101	163	122	168
False	182	162	239	182

performance of DepManthan as the False instances are clearly not the strength of DepManthan.

Table I represents the True and False instances solved by different DQBF solvers. DepManthan could solve 46 more True instances than DQBDD while generating *certificates*.

One interesting observation is, though, HQSPre as a preprocessor helps the state of the art DQBF solvers, strangely the performance of DepManthan degrades with HQSPre. To give a reference, DCAQE without HQSPre could solve *only* 233 instances, and DepManthan with HQSPre could solve merely 179 instances.

C. *Improvement in Virtual Best DQBF Solver*

The detailed analysis discussed in Section VII-B opens up an important question: *Does* DepManthan *improves the Virtual Best Solver (VBS) for DQBF?* VBS represents the union of DQBF solvers in an ideal portfolio. An instance is considered to be solved by VBS if is solved by any of the DQBF solver in the portfolio; that is, VBS is at least as powerful as each solver in the portfolio. The time taken to solve an instance by VBS is the minimum of time taken by the DQBF solvers to solve that instance.

Table II represents a pair wise comparison of DepManthan with other DQBF solvers. In Table II, the first row represents the number of instances that were not solved by DepManthan and solved by DQBF solver in corresponding column, and similarly, second row represents the number of instances that were solved by DepManthan and not solved by corresponding DQBF solver. Last column (All tools) of Table II represents union of DCAQE, HQS2 and DQBDD.

As presented in Table II, there are **67** instances that were solved by DepManthan and could not be solved by any of the other state of the art DQBF solvers — an addition of 67 instances in the number of instances solved by VBS.

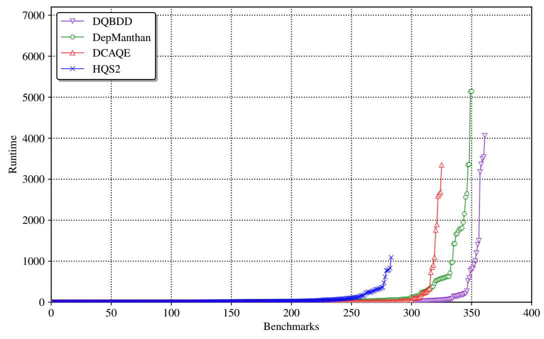

Fig. 7: DepManthan vs other DQBF solver.

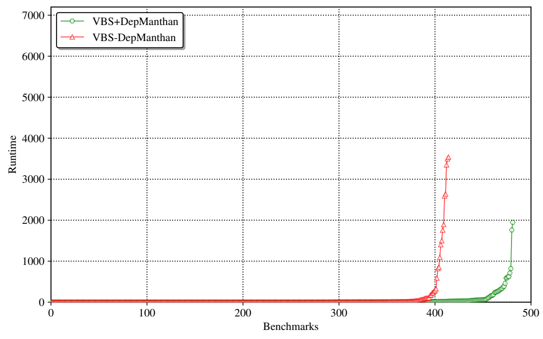

Fig. 8: VBS for DQBF solvering with/without DepManthan

TABLE II: DepManthan vs other state-of-the-art DQBF solvers

		HQS2	DCAQE	DQBDD	All tools
DepManthan	Less	53	116	95	131
	More	120	141	84	67

Figure 8 represents the cactus plot for VBS with and without DepManthan. Further analysis shows that DepManthan contributes 96 instances to the VBS which includes 67 new instances solved.

VIII. Conclusion

Henkin synthesis is an important problem in computer science with wide ranging applications. In this work, we purposed the first general purpose Henkin synthesiser, DepManthan which takes advantage from the advances in machine learning and automated reasoning. DepManthan not only showed significant performance improvement in synthesising Henkin functions, it also showed competitive performance as a DQBF solver.

References

[1] "QBF solver evaluation portal 2019." [Online]. Available: http://www.qbflib.org/qbfeval19.php

[2] "QBF solver evaluation portal 2020." [Online]. Available: http://www.qbflib.org/qbfeval20.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, "Whats hard about boolean functional synthesis?" in *Proc. of CAV*, 2018.

[6] V. Balabanov and J.-H. R. Jiang, "Resolution proofs and skolem functions in QBF evaluation and applications," in *Proc. of CAV*, 2011.

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

[8] R. Bloem, R. Könighofer, and M. Seidl, "Sat-based synthesis methods for safety specs," in *Proc. of VMCAI*, 2014.

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

[10] A. Fröhlich, G. Kovásznai, and A. Biere, "A dpll algorithm for solving dqbf," *Proc. POS*, 2012.

[11] A. Fröhlich, G. Kovásznai, A. Biere, and H. Veith, "idq: Instantiation-based dqbf solving." in *Proc. of SAT*, 2014.

[12] 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 ICCD*, 2013.

[13] K. Gitina, R. Wimmer, S. Reimer, M. Sauer, C. Scholl, and B. Becker, "Solving dqbf through quantifier elimination," in *Proc. of DATE*, 2015.

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

[15] L. Henkin, "Some remarks on infinitely long formulas, infinitistic methods," 1959.

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

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

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

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

[20] F. Lonsing and A. Biere, "DepQBF: A dependency-aware QBF solver," *Proc. of JSAT*, 2010.

[21] F. Lonsing and U. Egly, "Depqbf 6.0: A search-based QBF solver beyond traditional QCDCL," in *Proc. of CADE*, 2017.

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

[23] G. Peterson, J. Reif, and S. Azhar, "Lower bounds for multiplayer noncooperative games of incomplete information," *Computers & Mathematics with Applications*, 2001.

[24] J. R. Quinlan, "Induction of decision trees," *Proc. of Machine learning*, 1986.

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

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

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

[28] S. Sharma, S. Roy, M. Soos, and K. S. Meel, "Ganak: A scalable probabilistic exact model counter," in *Proc. IJCAI*, 2019.

[29] J. S, "Satisfiability of dqbf using binary decision diagrams," Master's thesis, 2020. [Online]. Available: https://is.muni.cz/th/prexv/

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

[31] L. Tentrup and M. N. Rabe, "Clausal abstraction for DQBF," in *Proc. of SAT*, 2019.

[32] K. Wimmer, R. Wimmer, C. Scholl, and B. Becker, "Skolem functions for dqbf," in *Proc. of ATVA*, 2016.

[33] R. Wimmer, S. Reimer, P. Marin, and B. Becker, "Hqspre–an effective preprocessor for qbf and dqbf," in *Proc. of TACAS*, 2017.