Boolean Functional Synthesis and its Applications

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

1 Problem Statement

Given a set of Boolean inputs and outputs and an underlying specification, the problem of functional synthesis is to synthesize outputs in terms of inputs such that the specification is met. Formally, let X be the set of inputs $x_1, \ldots, x_n$ and Y be the set of outputs $y_1, \ldots, y_m$, given a specification $\varphi(X, Y)$, synthesise a function vector $\boldsymbol{F} : \langle f_1(X), \ldots, f_m(X)\rangle$ such that $\forall X(\exists Y \varphi(X, Y) \leftrightarrow \varphi(X, \boldsymbol{F(X)})$. Boolean functional synthesis is a fundamental problem whose origin traces back to Boole's seminal work, which was subsequently pursued with the focus of decidability by Lowenheim and Skolem. Boolean functional synthesis is also called Skolem functional synthesis in the literature, and it has wide-ranging applications, including QBF solving [29], automated program repair and synthesis [33], and cryptography [23].

From a complexity-theoretic perspective, there exists an instance for which Boolean functional synthesis must take super polynomial time [3]. However, there exists a diverse set of algorithmic techniques that handle practical real word instances to synthesize Skolem functions, including extracting functions from proof of the validity of $\forall X \exists Y \varphi(X, Y)$, knowledge complication based techniques [3, 2], and the usage of incremental determinization [29]. Although, the recent years have seen significant performance improvement in solving more instances, but scalability remains the holy grail.

2 Our Contributions

In this section, we give an overview of our contributions towards Boolean functional synthesis and its applications.

2.1 Manthan [10]

We proposed a data-driven based approach, called Manthan [10]. Manthan takes advantage of recent advances in machine learning, constrained sampling, and automated reasoning for efficient Skolem functional synthesis.

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

Doctoral Program at 28th International Conference on Principles and Practice of Constraint Programming (CP 2022).
Editor: Hélène Verhaeghe; Article No. X; pp. X:1–X:9

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

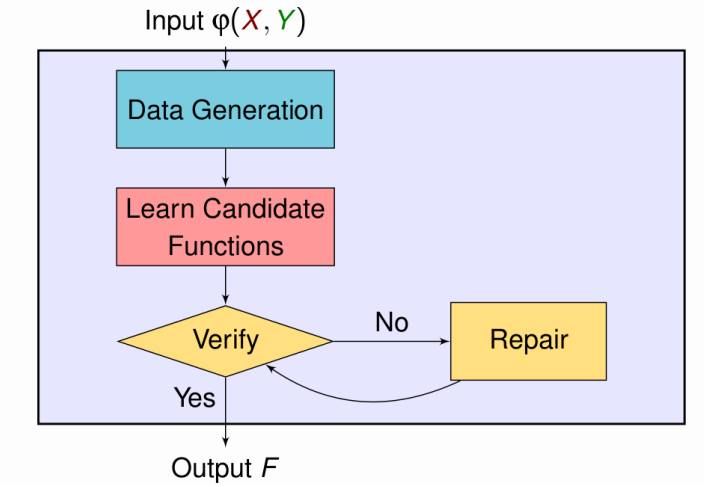

Figure 1 Manthan Framework

As the first step, Manthan samples the satisfying assignments of φ using an *a*daptative weighted sampling strategy using state-of-the-samplers [14, 13]. The generated samples would be considered as data to learn candidates in the later stages of Manthan.

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

The generated samples are used to learn an approximate candidate vector $\boldsymbol{F}$. The function, f_i, corresponding to each output variable f_i, is learned as a decision tree classifier. The valuations of X and Y in samples generated 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.

Proof-Guided Repair: Since machine learning techniques often produce good but inexact approximations, we augment our method with automated reasoning techniques to verify the correctness of decision tree-based candidate Skolem functions. To this end, we perform a counterexample driven refinement approach for candidate Skolem functions. To fully utilize the impressive test accuracy attained by machine learning models, we design a *proof-guided refinement* approach that seeks to identify and apply *minor* repairs to the candidate functions, in an iterative manner, until we converge to a provably correct Skolem function vector. In a departure from prior approaches utilizing the Shannon expansion and self-substitution, we first use a MaxSAT solver to determine potential repair candidates, and employ unsatisfiability cores obtained from the infeasibility proofs capturing the reason for current candidate functions to meet the specification, to construct

a *good repair*.
Essentially once, we have candidate vector $\boldsymbol{F}$, Manthan formally verifies whether $\boldsymbol{f}$ is the required Skolem function vector by doing a satisfiability check on formula $E(X, Y, Y') : \varphi(X, Y) \wedge \neg\varphi(X, Y') \wedge (Y' \leftrightarrow \boldsymbol{F})$ [3]. $E(X, Y, Y')$ formula is essentially checking if there exists a valuation of X for which $\exists Y \varphi(X, Y)$ evaluates to True, and $\varphi(X, \boldsymbol{F(X)})$ evaluates to false. Y' is fixed as per valuation of the output of current candidate functions; that is, y'_i is the same as $f_i(X)$. If $E(X, Y, Y')$ is UNSAT, the candidate function vector is a Skolem function vector, and Manthan returns $\boldsymbol{F}$, else we have a counterexample, and the candidates need to undergo a repair iterations to fix the counterexample. The candidate functions are repeatedly tested for correctness and repaired. Manthan uses a MaxSAT solver to minimize the number of repairs required for each counterexample, that is, it uses MaxSAT to identify a *good* counter examples. In the repair iterations, using an unsatcore based approach, Manthan attempts to find the *reason* why $\varphi(X, \boldsymbol{F(X)})$ evaluates to false along with counterexample valuation of X. Manthan then generates a repair formula using the unit clauses in the unsatcore to fix the counterexample.

We compare Manthan performance with the state of the art tools, viz. BFSS [3], C2syn [2], and CADET [29] on a set of benchmarks drawn from QBFEval-17-18 [1], Disjunctive, Factorization and Arithmetic data set [3]. Manthan significantly improves upon state of the art, and solves 356 benchmarks while the state of the art tool can only solve 280; in particular, Manthan solved 60 more benchmarks that could not be solved by any of the tools.

2.2 Manthan2 [12]

Although Manthan led to a significant improvement of the state-of-the-art, a large number of problems remain beyond its reach (and other synthesis engines). Therefore, we tackled the scalability challenge faced by Manthan. Specifically, we identify and address the following four key performance bottlenecks:

1. **Over-reliance on Data-driven Learning**: Manthan seeks to learn every function only based on samples and in turn, does not take full advantage of the white-box access to the formula $\varphi(X, Y)$.

   Remedy: We identify a subset of variables with *unique Skolem function* and extract these functions with an interpolantion-based technique, thereby reducing the number of functions that need to be learned.

2. **Inefficient Learning**: We observe that for some of the benchmarks, Manthan spends as much as 74% of its total running time on learning candidate functions from data.

   Remedy: Instead of relying solely on binary 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. **Under-usage of Determined Features**: While the conventional wisdom in formal methods is to perform variable elimination whenever possible, such an elimination robs the learning phase of *determined features*.

   Remedy: 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, retaining it instead as a possible feature during learning and repair.

4. **Dependency-Agnostic Repair**: Including variables from Y as features leads to dependencies among learned candidate functions. Manthan's use of MaxSAT queries in determining functions to be repaired fails to consider these dependencies, frequently resulting in an unnecessarily large number of repairs.

   Remedy: We propose the use of lexicographic MaxSAT for identifying repair candidates so as to take into account dependencies among candidate functions.

We implement these improvements in the framework called Manthan2. Manthan2 shows significantly improved runtime performance compared to Manthan. On the same 609 benchmarks, Manthan2 can synthesize 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.

We used Open-WBO [22] for unweighted MaxSAT queries, RC2 [16] for LexMaxSAT queries, and PicoSAT [7] to compute UnsatCore. Further, we used a library based on UNIQUE [32] to extract unique Skolem functions. Finally, we used Scikit-Learn [25] to learn decision trees and ABC [9] 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.

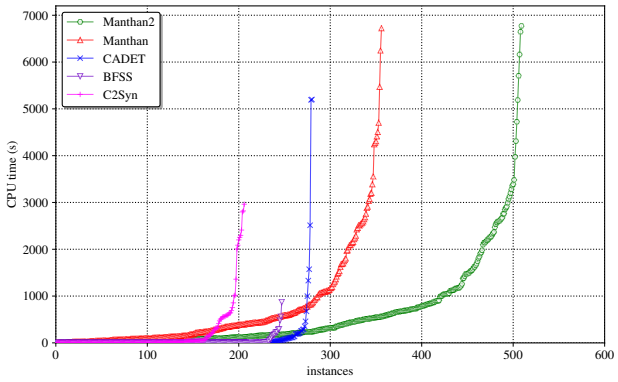

Figure 2 Manthan2 vs Manthan vs other state-of-the-art tools. Total instances 609. Timeout:7200 seconds

C2Syn [2]	BFSS [3]	CADET [29]	Manthan [10]	Manthan2 [12]
206	247	280	356	509

Table 1 Instances solved by each tool. Total instances 609. Timeout:7200 seconds

Figure 2 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 2, 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. Table 1 list the number of instances solved by each of the state-of-the-tools.

Encouraged by Manthan's scalability, we will seek to extend the above approach to related problem domains such as automated program synthesis, program repair, and reactive synthesis. In many functional synthesis applications like circuit repair, we need to find small size functions. Unfortunately, we are not aware of any techniques that attempt to synthesis the smallest possible Skolem functions. Coming up with a technique for Boolean functional synthesis with an objective of learning smallest sizes functions is an interesting direction for future work. Manthan is available open sourced at `https://github.com/meelgroup/manthan`.

2.3 Program Synthesis as DQF(T) [11]

The improvements in Skolem functional synthesis have paved the way for studying variants of functional, which are harder from a complexity-theoretic perspective. One such variant of interest is *Dependency Quantified Boolean Formulas (DQBF)*, which generalizes the well known notion of Quantified Boolean Formulas (QBF) $\forall X \exists Y \varphi(X, Y)$ by allowing explicit specification of dependency for existentially quantified variables, also known as *Henkin quantifiers* [15]. The expressiveness of DQBF comes at the cost of the hardness from a complexity-theoretic perspective: in particular, DQBF is NEXPTIME-complete [26].

The formula $\phi = \forall X \exists^{H_1} y_1 \exists^{H_2} y_2 \ldots \exists^{H_m} y_m \varphi(X, Y)$ is considered to be a DQBF formula, where each $H_i \subseteq \{x_1, \ldots, x_n\}$. H_i is considered as Henkin dependency corresponding to y_i. A DQBF formula is considered to be True if there exists a function vector $\boldsymbol{F} : \langle f_1(H_1), \ldots, f_m(H_m) \rangle$ such that $\varphi(X, \boldsymbol{F})$ is a tautology, otherwise DQBF is considered to be False. The problem of synthesizing such function vector $\boldsymbol{F}$ is considered as problem of Henkin synthesis. Henkin synthesis generalizes Skolem synthesis by allowing explicit specification of dependency for existentially quantified variables, also known as Henkin dependencies.

A crucial ingredient in the *NP revolution* was the reduction of key problems such as planning [17] and bounded model checking [8] to SAT. Such reductions served as a rich source of practical instances, and at the same time, planning and bounded model checking tools built on top of SAT achieved fruits of the progress in SAT solving and thereby leading to even wider adoption, and contributing to a virtuous cycle [21]. Our investigation in this work is in a similar spirit. The past few years have seen a surge of interest from diverse viewpoints such as the development of Dependency Quantified Boolean Formulas (DQBF) proof systems, the study of restricted fragments to development of efficient DQBF solvers [20, 29, 34]. In this work, we focus on a key problem in programming languages, program synthesis, and investigate its relationship to DQBF and its generalization, Dependency Quantified Formulas modulo Theory, henceforth referred to as DQF($\mathbb{T}$). Given a specification $\varphi(X, Y)$ over the set of inputs X and the set of outputs Y, the problem of program synthesis is to synthesize a program f such that $Y = f(X)$ would satisfy the specification φ.

The earliest work on synthesis dates back to Church [18], and the computational intractability of the problem defied development of practical techniques. A significant breakthrough was achieved with the introduction of Syntax-Guided Synthesis (SyGuS) formulation wherein in addition to φ, the input also contains a grammar of allowed implementations of f. The grammar helps to constrain the space of allowed implementation of f, and therefore, it also allows development of techniques that can efficiently enumerate over the grammar. While grammar has also been used as an implicit specification tool for few selected applications, it is mainly used to aid the underlying solver by constraining the search space. [4, 5, 6]. Often, the end user is primarily concerned with any function that can be expressed using a particular theory $\mathbb{T}$. For the sake of clarity, we use the term $\mathbb{T}$-*constrained* synthesis to characterize such class of synthesis problems. $\mathbb{T}$-constrained synthesis is a subclass of SyGuS.

Of particular interest is the recent work in the development of specialized algorithms focused on $\mathbb{T}$-constrained synthesis, e.g., counterexample-guided quantifier instantiation algorithm in [30]. Secondly, recent studies have also highlighted that for a wide variety of applications, the usage of grammar is solely for the purpose of aiding solver efficiency, and as such have advocated usage of more expressive grammars for a given SyGuS instance [24].

The primary contribution of our work is establishing a connection between Theory-constrained synthesis and DQF($\mathbb{T}$). In particular, our work makes the following contributions:

From $\mathbb{T}$-constrained synthesis to DQF($\mathbb{T}$): We present an reduction of $\mathbb{T}$-constrained synthesis to DQF($\mathbb{T}$). DQF($\mathbb{T}$) lifts the notion of DQBF from the Boolean domain to general Theory $\mathbb{T}$. We view the simplicity of the reduction from $\mathbb{T}$-constrained synthesis to DQF($\mathbb{T}$) as a core strength of the proposed approach.

Efficient $\mathbb{T}$-constrained synthesizers for $\mathbb{T}$=bitvectors: The reduction to DQF($\mathbb{T}$) opens up new directions for further work. As a first step, we focus on the case when the $\mathbb{T}$ is restricted to bitvector theory, denoted by BV. We observe that the resulting DQF(BV) instances can be equivalently specified as a DQBF instance. We demonstrate that our reduction to DQBF allows us to simply plug-in the state of the art DQBF solvers [34, 31].

Let us know bring our attention towards the reduction of T-constrained synthesis to DQF($\mathbb{T}$). A key strength of the reduction is its simplicity. Algorithm 1 formalizes the desired reduction. Before discussing the details of Algorithm 1, let us define the notation of $\mathsf{CallSigns}$. We refer to the function and its (ordered) list of arguments at an invocation (within φ) as its *call signature*. The set of all call signatures of a function symbol f in φ is referred by $\mathsf{CallSigns}(f)$. Note that the number of invocations of a function may not match $|\mathsf{CallSigns}(f)|$. For example, the following formula $\varphi : \forall a, b, c\ \exists f\ f(a, b) \wedge f(b, c) \wedge f(b, a) \wedge f(a, b)$, has 4 invocations of f while $\mathsf{CallSigns}(\mathrm{f}) = \{\langle a, b\rangle, \langle b, c\rangle, \langle b, a\rangle\}$. Note that $\langle a, b\rangle$ and $\langle b, a\rangle$ are considered as two different $\mathsf{CallSigns}$ of f.

In Algorithm 1 the reduction of φ to DQF($\mathbb{T}$) formulation is discussed, where φ is a specification over the vocabulary of background theory $\mathbb{T}$ with a set of typed function symbols $\{f_1, f_2, \ldots f_m\}$ such that for all f_i, $|\mathsf{CallSigns}(f_i)| = 1$. The important point to note is that the Henkin quantifiers must be carefully constructed so that each f_i depends only on the set of variables that appear in its argument-list.

Algorithm 1 Reducing single-callsign instance φ to DQF($\mathbb{T}$)

Input: A background theory $\mathbb{T}$, a set of typed function symbols $\{f_1, f_2, \ldots f_m\}$, a specification φ over the vocabulary of $\mathbb{T}$

1 Let $X = \bigcup_{f_i} \{h \mid h \in \mathsf{CallSigns}(f_i)\}$

2 Substitute every invocation of f_i with a fresh variable y_i in φ

3 Define $H_i = Set(h)$ as $\{h | h \in \mathsf{CallSigns}(f_i)\}$

Output: $\forall X \exists^{H_1} y_1.\ \exists^{H_2} y_2\ \ldots \exists^{H_m} y_m \varphi(X, Y)$

Now, let us turn our attention to the case when there exist a function f_i such that $|\mathsf{CallSigns}(f_i)| > 1$. In such cases, we pursue a Ackermannization-style technique that transforms φ into another specification $\hat{\varphi}$ such that every function f_i in $\hat{\varphi}$ has $|\mathsf{CallSigns}(f_i)| = 1$ (Algorithm 2). Note that this transformation allows the subsequent use of Algorithm 1 with $\hat{\varphi}$ to complete the reduction to DQF($\mathbb{T}$). The proposed transformations in Algorithm 2 are linear in the size of the formula like the transformation introduced in [27], however Algorithm 2 introduces lesser number of new variables.

The essence of Algorithm 2 is captured in the following two transformations:

Algorithm 2 Reducing multiple-callsign to single-callsign instance

Input: A background theory $\mathbb{T}$, a set of typed function symbols $\{f_1, f_2, \dots f_m\}$, a specification φ over the vocabulary of $\mathbb{T}$ such that $\ell_i = |\mathsf{CallSigns}(f_i)|$

1 **for** $i = 1$ *to* m **do**
2 **if** $|\mathsf{CallSigns}(f_i)| > 1$ **then**
3 Add a fresh (ordered) set of variables Z_i such that $|Z_i| = |\mathsf{CallSigns}(f_i)[0]|$
4 **for** $j \in [0 \dots (\ell_i - 1)]$ **do**
5 Replace every f_i whose $args(f_i) = \mathsf{CallSigns}(f_i)[j]$ with f_i^j
6 Add constraint $(args(f_i^j) = Z_i) \rightarrow f_i^j(args) = f_i^{\ell_i}(Z_i)$ to φ
7 **end**
8 $\mathsf{CallSigns}(f_i) \leftarrow \mathsf{CallSigns}(f_i) \cup \{Z_i\}$
9 **end**
10 **end**

Output: A set of typed function symbols $\{f_1^0, f_1^2, \dots f_1^{\ell_1}, \dots, f_m^0 \dots f_m^{\ell_m}\}$, a specification $\hat{\varphi}$ over the vocabulary of $\mathbb{T}$ such that $\forall i, j$ we have $|\mathsf{CallSigns}(f_i^j)| = 1$

(Line 5) We substitute instances of every call signature of f_i with fresh function symbols f_i^j (that corresponds to the j^{th} call signature of f_i). This reduces the formula from multiple-callsign to a single-callsign instance.

(Line 6) Introduction of an additional constraint for each f_i that forces all the functions f_i^j (introduced above) to mutually agree on every possible instantiation of arguments. Specifically, it introduces a fresh function symbol $f_i^{l_i}$ and a set of fresh variables $z_1^i, \dots, z_n^i \in Z_i$ such that, for all $args(f_i^j)$ argument lists, $(args(f_i^j) = Z_i) \implies f_i^j(args) = f_i^{\ell_i}(Z_i)$, where $j \in [0 \dots l_{i-1}]$.

When $\mathbb{T}$ is bitvector (BV): When the specification $\varphi(X, Y)$ is in BV. If $\varphi(X, Y)$ is a multiple-callsign instance, we use Algorithm 2 to covert it to a single-callsign instance $\hat{\varphi}(\hat{X}, \hat{Y})$. We then use Algorithm 1 to generate the DQF(BV) instance of $\hat{\varphi}(\hat{X}, \hat{Y})$ as $\forall \hat{X} \exists^{H_1} \hat{y}_1. \dots \exists^{H_m} \hat{y}_m \hat{\varphi}(\hat{X}, \hat{Y})$. Finally, we solve the DQF(BV) instance by compiling it down to a DQBF instance, thereby allowing the use of off-the-shelf DQBF solvers.

As the first step to DQBF compilation, we perform *bit-blasting* over $\hat{\varphi}$ to obtain $\hat{\varphi}'$.

$$\forall \hat{X} \exists^{H_1} \hat{y}_1 \dots \exists^{H_m} \hat{y}_m \hat{\varphi}(\hat{X}, \hat{Y}) \equiv \forall X' \exists^{X'} V. \exists^{H_1'} Y_1' \dots \exists^{H_m'} Y_m' \varphi'(X', Y') \quad (1)$$

The objective of our experimental evaluation was to study the feasibility of solving BV-constrained synthesis via the state-of-the-art DQBF solvers. To this end, we perform an evaluation over an extensive suite of 645 general-track bitvector (BV) theory benchmarks from SyGuS competition 2018, 2019. We used CVC4 [30], EUSolver [5], ESolver [35] as SyGuS-tools. CVC4 was also used in its BV-constrained version. We used state-of-the-art DQBF (QBF) solvers CADET [28], Manthan [10], DepQBF [19], DCAQE [34] and DQBDD [31].

Table 2 represents the instances solved by the virtual best solver for SyGuS, BV constrained, and DQBF tools.

Table 2 Number of SyGuS solved using different techniques. Timeout 900s.

	Total	SyGuS-tools	BV-constrained	DQBF-based
SyGuS Instances	645	513	606	610

As shown in Table 2, with syntax guided synthesis, we could synthesize the functions for 513 out of 645 SyGuS instances only, whereas, with BV-constrained synthesis, we could

solve 606 such instances. Surprisingly, BV-constrained synthesis performs better than the syntax-guided synthesis. Table 2 also shows that the DQBF based synthesis tools perform similar to BV-constrained synthesis tools for SyGuS instances; this provides strong evidence that the general purpose DQBF solvers can match the efficiency of the domain specific synthesis tools.

Syntax-guided synthesis has emerged as a dominant paradigm for program synthesis. Motivated by the impressive progress in automated reasoning, we investigate the usage of syntax as a tool to aid the underlying synthesis engine. To this end, we formalize the notion of $\mathbb{T}$-constrained synthesis, which can be reduced to DQF($\mathbb{T}$). We then focus on the special case when $\mathbb{T} = BV$. The corresponding BV-constrained synthesis can be reduced to DQBF, highlighting the importance of the scalability of DQBF solvers. Our empirical analysis shows that $\mathbb{T}$-constrained synthesis can achieve significantly better performance than syntax-guided approaches. Furthermore, the general purpose DQBF solvers perform on par with domain-specific synthesis techniques and thereby supporting the argument of viewing DQBF as a general purpose representation language for representation task. We believe that our results will motivate further research into DQBF; the rewards of which can be reaped by program synthesis tools. The tool to convert a program synthesis instances in bit vector theory to a DQBF formula is available open-sourced at `https://github.com/meelgroup/dequs`.

References

1 QBF solver evaluation portal 2018, 2018. URL: `http://www.qbflib.org/qbfeval18.php`.

2 S Akshay, Jatin Arora, Supratik Chakraborty, S Krishna, Divya Raghunathan, and Shetal Shah. Knowledge compilation for boolean functional synthesis. In *Proc. of FMCAD*, 2019.

3 S Akshay, Supratik Chakraborty, Shubham Goel, Sumith Kulal, and Shetal Shah. What's hard about boolean functional synthesis? In *Proc. of CAV*, 2018.

4 Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo MK Martin, Mukund Raghothaman, Sanjit A Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *Proc. of FMCAD*, 2013.

5 Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. Scaling enumerative program synthesis via divide and conquer. In *Proc. of TACAS*, 2017.

6 Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovi'c, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *Proc. of CAV*, 2011.

7 Armin Biere. PicoSAT essentials. *Proc. of JSAT*, 2008.

8 Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without bdds. In *International conference on tools and algorithms for the construction and analysis of systems*, pages 193–207. Springer, 1999.

9 Robert Brayton and Alan Mishchenko. Abc: An academic industrial-strength verification tool. In *International Conference on Computer Aided Verification*, pages 24–40. Springer, 2010.

10 Priyanka Golia, Subhajit Roy, and Kuldeep S. Meel. Manthan: A data-driven approach for boolean function synthesis. In *Proceedings of International Conference on Computer-Aided Verification (CAV)*, 2020.

11 Priyanka Golia, Subhajit Roy, and Kuldeep S. Meel. Program synthesis as dependency quantified formula modulo theory. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, 2021.

12 Priyanka Golia, Friedrich Slivovsky, Subhajit Roy, and Kuldeep S. Meel. Engineering an efficient boolean functional synthesis engine. In *Proceedings of International Conference On Computer Aided Design (ICCAD)*, 2021.

13 Priyanka Golia, Mate Soos, Sourav Chakraborty, and Kuldeep S. Meel. Designing samplers is easy: The boon of testers. In *Proceedings of Formal Methods in Computer-Aided Design (FMCAD)*, 2021.

14 Rahul Gupta, Shubham Sharma, Subhajit Roy, and Kuldeep S Meel. WAPS: Weighted and projected sampling. In *Proc. of TACAS*, 2019.

15 Leon Henkin. Some remarks on infinitely long formulas, infinitistic methods (proc. sympos. foundations of math., warsaw, 1959), 1961.

16 Alexey Ignatiev, Antonio Morgado, and Joao Marques-Silva. PySAT: A Python toolkit for prototyping with SAT oracles. In *Proc. of SAT*, 2018. URL: `https://doi.org/10.1007/978-3-319-94144-8_26`.

17 Henry A Kautz and Bart Selman. Planning as satisfiability. In *Proc. of ECAI*, 1992.

18 Andrei N Kolmogorov. „zur deutung der intuitionistischen logik mathematische zeitschrift35. *English translation in VM Tikhomirov (ed.) Selected Works of AN Kolmogorov*, 1932.

19 Florian Lonsing and Armin Biere. DepQBF: A dependency-aware QBF solver. *Proc. of JSAT*, 2010.

20 Florian Lonsing and Uwe Egly. Depqbf 6.0: A search-based QBF solver beyond traditional QCDCL. In *Proc. of CADE*, 2017.

21 Joao Marques-Silva, Inês Lynce, and Sharad Malik. Conflict-driven clause learning sat solvers. In *Handbook of satisfiability*, pages 131–153. ios Press, 2009.

22 Ruben Martins, Vasco Manquinho, and Inês Lynce. Open-WBO: A modular MaxSAT solver. In *Proc. of SAT*, 2014.

23 Fabio Massacci and Laura Marraro. Logical cryptanalysis as a SAT problem. *Journal of Automated Reasoning*, 2000.

24 Saswat Padhi, Todd D. Millstein, Aditya V. Nori, and Rahul Sharma. Overfitting in synthesis: Theory and practice. In *Proc. of CAV*, 2019.

25 Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *Journal of machine learning research*, 12(Oct):2825–2830, 2011.

26 Gary Peterson, John Reif, and Salman Azhar. Lower bounds for multiplayer noncooperative games of incomplete information. *Computers & Mathematics with Applications*, 2001.

27 Markus N Rabe. A resolution-style proof system for DQBF. In *Proc. of SAT*, 2017.

28 Markus N Rabe. Incremental determinization for quantifier elimination and functional synthesis. In *Proc. of CAV*, 2019.

29 Markus N Rabe, Leander Tentrup, Cameron Rasmussen, and Sanjit A Seshia. Understanding and extending incremental determinization for 2QBF. In *Proc. of CAV*, 2018.

30 Andrew Reynolds, Morgan Deters, Viktor Kuncak, Cesare Tinelli, and Clark Barrett. Counterexample-guided quantifier instantiation for synthesis in SMT. In *Proc. of CAV*, 2015.

31 Juraj Sičˇ. Satisfiability of DQBF using binary decision diagrams. Master's thesis, 2020. URL: `https://is.muni.cz/th/prexv/`.

32 Friedrich Slivovsky. Interpolation-based semantic gate extraction and its applications to QBF preprocessing. In *Proc. of CAV*, 2020.

33 Saurabh Srivastava, Sumit Gulwani, and Jeffrey S Foster. Template-based program verification and program synthesis. *STTT*, 2013.

34 Leander Tentrup and Markus N Rabe. Clausal abstraction for DQBF. In *Proc. of SAT*, 2019.

35 Abhishek Udupa, Arun Raghavan, Jyotirmoy V Deshmukh, Sela Mador-Haim, Milo MK Martin, and Rajeev Alur. Transit: specifying protocols with concolic snippets. *ACM SIGPLAN Notices*, 2013.