Program Synthesis as Dependency Quantified Formula Modulo Theory

Priyanka Golia\textsuperscript{1,2}, Subhajit Roy\textsuperscript{1}, Kuldeep S. Meel\textsuperscript{2}

\textsuperscript{1}Indian Institute of Technology Kanpur
\textsuperscript{2}National University of Singapore

Abstract
Given a specification $\phi(X, Y)$ over inputs $X$ and output $Y$, defined over a background theory $\mathbb{T}$, the problem of program synthesis is to design a program $f$ such that $Y = f(X)$ satisfies the specification $\phi$. Over the past decade, syntax-guided synthesis (SyGuS) has emerged as a dominant approach for program synthesis where in addition to the specification $\phi$, the end-user also specifies a grammar $L$ to aid the underlying synthesis engine. This paper investigates the feasibility of synthesis techniques without grammar, a sub-class defined as $\mathbb{T}$-constrained synthesis. We show that $\mathbb{T}$-constrained synthesis can be reduced to $\text{DQF}(\mathbb{T})$, i.e., to the problem of finding a witness of a Dependency Quantified Formula Modulo Theory. When the underlying theory is the theory of bitvectors, the corresponding $\text{DQF}(\text{BV})$ problem can be further reduced to Dependency Quantified Boolean Formulas (DQBF). We rely on the progress in DQBF solving to design DQBF-based synthesizers that outperform the domain-specific program synthesis techniques, thereby positioning DQBF as a core representation language for program synthesis. 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.

1 Introduction
The past three decades have been witness to the so-called NP revolution aided by the development of sophisticated heuristics for SAT solving leading to modern solvers being able to handle formulas involving millions of variables [Marques-Silva et al., 2009]. The progress in SAT solving has led to effort for problems that lie in complexity classes beyond NP such as Quantified Boolean Formulas (QBF) [Akshay et al., 2018; Lonsing and Egly, 2017; Rabe, 2019; Rabe et al., 2018], Max-SAT [Martins et al., 2014], model or MUS counting [Sharma et al., 2019; Soos et al., 2020; Bendik and Meel, 2020], and the like.

Of particular interest to us is the development of algorithmic approaches for QBF from theoretical and practical perspectives [Akshay et al., 2018; Lonsing and Biere, 2010; Lonsing and Egly, 2017; Rabe, 2019; Rabe and Tentrup, 2015; Rabe et al., 2018]. These improvements have paved the way for studying variants of QBF, which are harder from 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) by allowing explicit specification of dependency for existentially quantified variables, also known as Henkin quantifiers [Henkin, 1961]. The expressiveness of DQBF comes at the cost of the hardness from a complexity-theoretic perspective: in particular, DQBF is NEXPTIME-complete [Peterson et al., 2001]. Nevertheless, as noted by Ganian et al. [Ganian et al., 2020], motivated by the progress in QBF solving, the past few years have seen a surge of interest from diverse viewpoints such as the development of DQBF proof systems, the study of restricted fragments to development of efficient DQBF solvers [Lonsing and Egly, 2017; Rabe et al., 2018; Tentrup and Rabe, 2019]. In this work, we will focus on DQBF and its generalization, Dependency Quantified Formulas modulo Theory, henceforth referred to as $\text{DQF}(\mathbb{T})$.

A crucial ingredient in the NP revolution was the reduction of key problems such as planning [Kautz and Selman, 1992] and bounded model checking [Biere et al., 1999] 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 [Marques-Silva et al., 2009].

Our investigation in this paper is in a similar spirit: we focus on a key problem in programming languages, program synthesis, and investigate its relationship to $\text{DQF}(\mathbb{T})$ and DQBF. Given a specification $\phi(X, Y)$ 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 $\phi$. The earliest work on synthesis dates back to Church [Kolmogorov, 1932], and the computational intractability of the problem defied development of practical techniques. A significant breakthrough was

\textsuperscript{*}The open source tool DeQuS based on this work is available at https://github.com/meelgroup/DeQuS

1894
achieved with the introduction of Syntax-Guided Synthesis (SyGuS) [Alur et al., 2013] formulation wherein in addition to \( \varphi \), 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. [Alur et al., 2013; Alur et al., 2017; Burrett et al., 2011]. 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\(^1\) to characterize such class of synthesis problems. Two observations are in order: first, \( \mathbb{T} \)-constrained synthesis is a subclass of SyGuS, i.e., every instance of \( \mathbb{T} \)-constrained synthesis is also an instance 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 [Reynolds et al., 2015]. 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 [Padhi et al., 2019].

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 efficient 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 DQBF(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 [Fröhlich et al., 2014; Gitina et al., 2015; Rabe and Tentrup, 2015; Sic, 2020].

The Power of DQBF. The remarkable progress in DQBF over the past few years is evident in our observation that DQBF-based synthesizers perform significantly better than domain-specific techniques that focus on utilization of the grammar to achieve efficiency. Our investigations were motivated by the classical work of Kautz and Selman [1992] that showcased the power of general purpose SAT solvers in achieving advances over domain specific techniques. In this spirit, our work showcases the power of reducing synthesis to DQF(\( \mathbb{T} \)). One question that we did not address so far is the choice of DQF(\( \mathbb{T} \)) over \( \mathbb{T} \)-constrained synthesis. To this end, we again focused on the case when \( \mathbb{T} \)=BV, and we transform DQBF benchmarks to program synthesis, and perform a comparison of program synthesis techniques vis-a-vis DQBF techniques. We observe that DQBF techniques significantly outperform the program synthesis techniques for these benchmarks; these results highlight the versatility of DQBF techniques and provide evidence in support of our choice of DQBF as the representation language.

**Role of Grammars.** Since DQBF-based synthesis techniques perform better than techniques that rely on grammar for efficiency, we would like to put forth the thesis of the usage of grammar as a specification tool rather than to guide the search strategy of the underlying synthesis engines, i.e., evolution of syntax-guided synthesis to syntax-constrained synthesis.

### 2 Notation and Preliminaries

We will focus on the typed first order logic formulas associated with a background theory \( \mathbb{T} \). We assume (without loss of generality) that function arguments are variables\(^2\). We refer to the function and its (ordered) list of arguments at an invocation (within \( \varphi \)) as its call signature. The set of all call signatures of a function symbol \( f \) in \( \varphi \) is referred by \( \text{CallSigns}(f) \). Note that the number of invocations of a function may not match \( |\text{CallSigns}(f)| \). For example, the following formula \( \varphi \), has 4 invocations of \( f \) while \( \text{CallSigns}(f) = \{(a,b), (b,c), (b,a)\} \). Note that \((a,b) \) and \((b,a) \) are considered as two different CallSigns of \( f \).

\[
\varphi : \forall a,b,c \exists f f(a,b) \land f(b,c) \land f(b,a) \land f(a,b)
\]

If every invocation of \( f \) in \( \varphi \) has an identical argument-list, then \( |\text{CallSigns}(f)| = 1 \), and we refer to \( \varphi \) as a single-callsignature instance. Otherwise, \( \varphi \) is a multiple-callsignature instance. We use \( \text{args}(f) \) to get the argument lists of function \( f \), and \( f(\text{args}) \) to represents the invocation of \( f \) with its args. While from the point of view of expressiveness, multiple functions can be reduced to the case of a single function, the performance of synthesis tools is affected by the number of functions to be synthesized.

We use BV for the bitvector theory. We use the lower case letters with subscripts for a variable, and the upper case letters for the set of variables. A literal is either a variable or its negation, and a clause is a disjunction of literals. A conjunctive normal form (CNF) formula is a conjunction of clauses.

We say that \( \Phi \) is a Dependency-Quantified Boolean Formula (DQBF) if \( \Phi \) can be represented as:

\[
\forall X. \exists H, y_1, \ldots, \exists H, y_m. \varphi(X,Y),
\]

where \( \varphi \) is a quantifier-free propositional formula and \( H_i \subseteq X \) such that the variables \( y_i \) can depend only on the variables of \( H_i \). We refer to \( \exists H \) as a Henkin quantifier. We also use the notation \( \exists H V \) to represent that every variable of the set \( V \) can depend only on the variables of \( H \). If \( \forall i \in [m] \forall H_i = X \), then \( \Phi \) is considered as a 2-Quantified Boolean Formula (2-QBF).

\(^1\)defined formally in Section 2

\(^2\)we can achieve such a representation by simply replacing every term \( t \) in the argument of a function \( f \) with a variable \( x \) and conjuncting \( \varphi \) with a constraint \( x = t \)
Henkin functional synthesis. For a given Boolean relational specification \( \exists H_1 y_1, \ldots, \exists H_m y_m, \varphi(X,Y) \) where \( X = \{x_1, \ldots, x_n\}, Y = \{y_1, \ldots, y_m\} \) and each \( H_i \subseteq X \), synthesize a function vector \( f = (f_1(H_1), \ldots, f_m(H_m)) \), such that \( y_i \leftrightarrow f_i(H_i) \) and \( \exists H_1 y_1, \exists H_2 y_2, \ldots, \exists H_m y_m, \varphi(X,Y) \equiv \varphi(X,f) \). The function vector \( f \) is also called Henkin function vector and each \( f_i \) is called Henkin function.

3 Related Work

From a theoretical perspective, the satisfiability problem of Dependency Quantified Boolean formulas (DQBF) is known to be NEXPTIME-complete [Peterson et al., 2001]. Despite the theoretical hardness, the past decades have been witness to substantial development in both QBF and DQBF solving [Rabe, 2019; Rabe and Tentrup, 2015; Sič, 2020; Tentrup and Rabe, 2019]. Frohlich et al. [2012] proposed the first DPLL based algorithm to lift the QBF solving to DQBF; Tentrup and Rabe [2019] followed the same direction, and proposed the idea of using clausal abstraction for DQBF solving. In terms of QBF solvers, the series of work by Rabe et al. [2019; 2015] showed that the idea of using incremental determination not only improves the state-of-the-art for QBF solving, but also can be used for Skolem functional synthesis. In a separate line of work Jiang [2009] introduced Craig interpolation-based Skolem function synthesis approach. Subsequently, CEGR-based approaches have also been proposed [Akshay et al., 2019; Akshay et al., 2018]. Fried, Tabajara and Vardi [2016] combined RBDDBased approach with CEGR for Skolem function synthesis. Recently, Golia, Roy and Meel [2020] proposed a data-driven approach for Skolem functional synthesis.

The origins of program synthesis trace back to 1930s. The work on constructive mathematics [Kolmogorov, 1932] initiated the idea of constructing interpretable solutions with proofs by composing solutions of smaller sub-problems, followed up by deductive synthesis strategies [Green, 1981; Manna and Waldinger, 1971]. Alur et al. [2013] proposed the idea of using grammar for syntax-guided synthesis: they showed that using grammar would result in efficient program synthesis with more interpretable programs [Alur et al., 2013; Alur et al., 2017; Udupa et al., 2013]. Another line of work is to build the synthesizer on top of SAT/SMT solver [Reynolds et al., 2015]. CVC4 [Reynolds et al., 2015] is the first synthesis engine implemented inside an SMT solver, it extracts the desired functions from the unsatisfiability proofs of the negated form of synthesis conjectures.

In recent years, there has been work on understanding the impact of provided grammar on the performance of existing tools. Padhi et al. [2019] empirically showed that with the increase of expressiveness of the provided grammar, the performance of the tools deteriorates significantly. Recently, Kim et al. [2021] introduced semantic guided synthesis, where the user is allowed to specify both syntax and semantics.

Prior works have identified the use of DQBF solving for reactive and bounded synthesis [Faymonville et al., 2017; Gitina et al., 2015; Tentrup and Rabe, 2019]. Tentrup and Rabe [2019] demonstrates that the usage of clausal abstraction for DQBF leads to performance improvements for the benchmarks arising from reactive synthesis. In this context, our reduction from program synthesis for BV programs to DQBF further reinforces the centrality of DQBF as a representation language.

4 Synthesis via DQF(\( \mathbb{T} \))

Let us first discuss the well-known formula of syntax-guided synthesis (SyGuS). In this formulation, the constraints over the functions to be synthesized are specified in the vocabulary of a given background theory \( \mathbb{T} \) along with the function symbols. Notice that the background theory specifies the domain of values for each variable type along with the interpretation for the function(s) and predicate symbols in the vocabulary.

**Definition 1** ([Alur et al., 2013]). Given a background theory \( \mathbb{T} \), a set of typed function symbols \( \{f_1, f_2, \ldots, f_k\} \), a specification \( \varphi \) over the vocabulary of \( \mathbb{T} \), a set of expressions \( \{L_1, \ldots, L_k\} \) over the vocabulary of \( \mathbb{T} \) such that \( L_i \) is of the same type as \( f_i \), the problem of syntax-guided synthesis (SyGuS) is to find a set of expressions \( \{e_1 \in L_1, e_2 \in L_2, \ldots, e_k \in L_k\} \) such that the formula \( \varphi[f_1/e_1, f_2/e_2, \ldots, f_k/e_k] \) is valid modulo \( \mathbb{T} \). Note that \( \varphi[f_1/e_1] \) denotes the result of substitution of \( f_1 \) with expression \( e_1 \), such that the bindings of inputs to \( f_1 \) is ensured.

In this work, we are interested in the subclass of SyGuS where \( L_i \) corresponds to the complete vocabulary of \( \mathbb{T} \); we call such a class as T-constrained synthesis, defined formally below:

**Definition 2.** Given a background theory \( \mathbb{T} \), a set of typed function symbols \( \{f_1, f_2, \ldots, f_k\} \), a specification \( \varphi \) over the vocabulary of \( \mathbb{T} \), a set of expressions \( \{e_1, e_2, \ldots, e_k\} \) defined over vocabulary of \( \mathbb{T} \) such that the formula \( \varphi[f_1/e_1, f_2/e_2, \ldots, f_k/e_k] \) is valid modulo \( \mathbb{T} \).

We propose the reduction of T-constrained synthesis to DQF(\( \mathbb{T} \)), i.e, to the problem of finding a witness of a dependency quantified formula modulo theory.

4.1 T-Constrained Synthesis to DQF(\( \mathbb{T} \))

As remarked in Section 1, a key strength of the reduction is its simplicity. Algorithm 1 formalizes the desired reduction of \( \varphi \) to DQF(\( \mathbb{T} \)) formulation where \( \varphi \) 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 \( f_i \), \( |\text{CallSigns}(f_i)| = 1 \). 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.

Now, let us turn our attention to the case when there exist \( f_i \) such that \( |\text{CallSigns}(f_i)| > 1 \). In such cases, we pursue a Ackermannization-style technique that transforms \( \varphi \) into another specification \( \tilde{\varphi} \) such that every function \( f_i \) in \( \tilde{\varphi} \) has \( |\text{CallSigns}(f_i)| = 1 \) (Algorithm 2). Note that this transformation allows the subsequent use of Algorithm 1 with \( \tilde{\varphi} \) to complete the reduction to DQF(\( \mathbb{T} \)). The proposed transformations in Algorithm 2 are linear in the size of the formula
As the formula on the right-hand side in Eq. 1 is an instance of DQBF, we can simply invoke an off-the-shelf certifying DQBF solvers to generate the Henkin functions for \( X' \).

A careful reader will observe that the Henkin functions corresponding to \( y'_i \) variables will be constructed in the propositional theory and not in BV, but note that one can simply convert a formula in propositional theory into an equivalent formula in bitvector theory defined over \( \hat{X}, \hat{y}_1, \ldots, \hat{y}_m \) with only a linear size increase in the representation.

While theoretical analysis is not the focus of this work, we observe that our reduction to DQBF allows one to lift the recently obtained results for DQBF to obtain Fixed Parameter Tractability (FPT) analysis for synthesis [Ganian et al., 2020]. We leave a detailed theoretical analysis to future work.

5 Experimental Evaluation

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 benchmarks and tools, which we describe in detail below.

5.1 Experimental Setup

Tools under Evaluation

Given a program synthesis instance defined over BV, we sought to compare three different possibilities: executing a SyGuS tool, executing a BV-constrained synthesis, and a DQBF-augmented synthesis framework. To this end, we experimented with the following tools (Table 1):

**Syntax-guided synthesis.** SyGuS tools for bitvector theory spanning symbolic, stochastic and enumerative solvers, like CVC4 [Barrett et al., 2011], EUSolver [Alur et al., 2017], ESolver [Udupa et al., 2013], Stochpp [Alur et al., 2013], Symbolic Solver [Alur et al., 2013], DryadSymbHuang et al., 2020]. Since, we employ multiple versions of CVC4, we will refer to the SyGuS-based variant of CVC4 as CVC4_{enum}. BV-constrained synthesis. Observe that every SyGuS tool can be transformed into a theory-constrained tool by a simple pipeline that rewrites the grammar of the input instance to be the entire vocabulary of the corresponding theory. Our empirical evaluation, however, revealed that such a transformation is ineffective, and the tools under transformation consistently perform worse than their corresponding versions under syntax-guided mode. In addition to the SyGuS-based BV-constrained tools, we employ the state of the art synthesis engine CVC4 [Reynolds et al., 2015] under the counter-example-guided quantifier instantiation mode, which can be viewed as a native approach to BV-constrained synthesis. We refer CVC4_{ex} to denote CVC4 invoked with counterexample-guided quantifier instantiation.

**DQBF-based synthesis.** The set of underlying DQBF solvers that we have employed in DQBF-based synthesis framework span CADET [Rabe, 2019], Manthan [Golia et al., 2020], DepQBF [Lonsing and Biere, 2010], DCAQE [Tentrup and Rabe, 2019] and DQBDD [Sic, 2020]. A careful reader might observe that CADET and Manthan are QBF solvers, i.e., they can only handle the special case when the

---

**Algorithm 1:** Reducing a single-callsign instance \( \varphi \) 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 \( \varphi \) over the vocabulary of \( \mathbb{T} \)

1. Let \( X = \bigcup_{f_i} \{h \mid h \in \text{CallSigns}(f_i)\} \)
2. Substitute every invocation of \( f_i \) with a fresh variable \( y_i \) in \( \varphi \)
3. Define \( H_i = \text{Set}(h) \{h \mid h \in \text{CallSigns}(f_i)\} \)

**Output:** \( \forall X \exists H_1 y_1, \exists H_2 y_2 \ldots \exists H_m y_m \varphi(X, Y) \)

---

like the transformation introduced in [Rabe, 2017], however Algorithm 2 introduces lesser number of new variables.

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

**(Line 5.)** We substitute instances of every call signature of \( f_i \) with fresh function symbols \( f'^i_i \) (that corresponds to the \( j \)-th call signature of \( f_i \)). This reduces the formula from multi-callsign to a single-callsign instance.

**(Line 6.)** Introduction of an additional constraint for each \( f_i \) that forces all the functions \( f'^i_i \) and a set of fresh variables \( z'_1, \ldots, z'_n \in Z_i \) such that, for all argument lists \( \text{args}(f'^i_i) \), we have \( \text{args}(f'^i_i) = Z_i \implies f'^i_i(\text{args}) = f'^i_i(Z_i) \) where \( j \in \{0, \ldots, l_i - 1\} \).
SyGuS competition and QBF competition. We use
The benchmark suite consisted of instances from two sources:
Benchmarks
existentially quantified variables are allowed to depend on all

5.2 Objectives and Summary
The primary objective of empirical evaluation is to show how
general purpose DQBF techniques can be transformed to ef-
fective synthesis engines. In particular, our empirical evalua-
tion sought answers to the following questions:

RQ1: Utility of Syntax for Efficiency. How does theory
constrained synthesis compare with syntax-guided synthesis?

RQ2: Domain Specific vs DQBF. Can general purpose
DQBF solver-based synthesis framework match the
efficiency of domain specific synthesis tools?

RQ3: Efficacy of DQBF as Representation Language.
Are BV-constrained synthesis an efficient approach to
DQBF solving?

Summary of Results
Table 2 represents the instances solved by the virtual best
solver for SyGuS, BV constrained, and DQBF tools. The first
row of Table 2 lists the number of instances solved by differ-
sent synthesis techniques: SyGuS based, BV-constrained syn-
thesis, and DQBF based synthesis for SyGuS instances, and
the second row represents the same for DQBF instances.

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 pur-
pose DQBF solvers can match the efficiency of the domain
specific synthesis tools. Furthermore, BV-constrained syn-
thesis tools perform poorly with DQBF as representation lan-
guage providing support for the efficacy of DQBF as a rep-
sentation language.

Table 2: Number of SyGuS and DQBF instances solved using dif-
erent techniques. Timeout 900s.

<table>
<thead>
<tr>
<th></th>
<th>Total</th>
<th>SyGuS-based tools</th>
<th>BV-constrained tools</th>
<th>DQBF-based tools</th>
</tr>
</thead>
<tbody>
<tr>
<td>SyGuS</td>
<td>645</td>
<td>513</td>
<td>606</td>
<td>610</td>
</tr>
<tr>
<td>DQBF</td>
<td>609</td>
<td>-</td>
<td>2</td>
<td>276</td>
</tr>
<tr>
<td>All</td>
<td>1254</td>
<td>513</td>
<td>608</td>
<td>886</td>
</tr>
</tbody>
</table>

Table 1: Tools used in our evaluation

<table>
<thead>
<tr>
<th></th>
<th>Syntax-Guided</th>
<th>BV-Constrained</th>
<th>DQBF-based</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVC4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>enum.</td>
<td>ESolver</td>
<td>CADET, DCAQE,</td>
<td></td>
</tr>
<tr>
<td>ESolver</td>
<td>EUSolver</td>
<td>Manthan,</td>
<td></td>
</tr>
<tr>
<td>DryadSynth</td>
<td></td>
<td>DepQBF,</td>
<td></td>
</tr>
<tr>
<td>Stochpp</td>
<td></td>
<td>DQBDD</td>
<td></td>
</tr>
</tbody>
</table>

Benchmark:
The benchmark suite consisted of instances from two sources:
SyGuS competition and QBF competition. We use 645
general-track bitvector (BV) theory benchmarks from SyGuS
competition 2018, 2019 to evaluate the performance of Sy-
GuS tools. To employ our DQBF-based framework, we used
Z3 [De Moura and Bjørner, 2008] to convert the instances
from SyGuS to DQBF. Furthermore, we considered 609 QBF
benchmarks from QBFeval competition 17, 18, disjunctive
decomposition and arithmetic set [Akshay et al., 2019; Rabe,
2019] and converted them to SyGuS instances. We consid-
ered each propositional variable as a bitvector of size 1, and
allow the synthesized function to depend on all the univer-
sally quantified variables. The associated grammar for these
benchmarks is the entire BV-vocabulary.

Implementation and Setup
Our reduction is implemented in a tool, DeQuS. All our ex-
periments were conducted on a high-performance computer
cluster with each node consisting of an E5-2690 v3 CPU with
24 cores and 96GB of RAM, with a memory limit set to 4GB
per core. All tools were run in a single-threaded mode on a
single core with a timeout of 900s.

5.2 Objectives and Summary
The primary objective of empirical evaluation is to show how
general purpose DQBF techniques can be transformed to ef-

\[ \text{Algorithm 2: Reducing multiple-callsign to single-callsign instance} \]
\[
\textbf{Input:} \text{A background theory } T, \text{ a set of typed function symbols } \{f_1, f_2, \ldots, f_m\}, \text{ a specification } \varphi \text{ over the vocabulary of } T \text{ such that } \ell_i = |\text{CallSigns}(f_i)|
\]
\[\begin{align*}
\text{for } i & = 1 \text{ to } m \text{ do} \\
\text{if } |\text{CallSigns}(f_i)| > 1 & \text{ then} \\
\text{Add a fresh (ordered) set of variables } Z_i \text{ such that } |Z_i| = |\text{CallSigns}(f_i)[0]| \\
\text{for } j & \in [0 \ldots (\ell_i - 1)] \text{ do} \\
\text{Replace every } f_i \text{ whose } \text{args}(f_i) = \text{CallSigns}(f_i)[j] \text{ with } f_i^j \\
\text{Add constraint } (\text{args}(f_i) = Z_i) & \rightarrow f_i^j(\text{args}) = f_i^j(Z_i) \text{ to } \varphi \\
\text{CallSigns}(f_i) & \leftarrow \text{CallSigns}(f_i) \cup \{Z_i\}
\end{align*}\]
\[
\textbf{Output:} \text{A set of typed function symbols } \{f_1^0, f_1^1, \ldots, f_1^{\ell_1}, \ldots, f_m^0, f_m^{\ell_m}\}, \text{ a specification } \varphi \text{ over the vocabulary of } T \text{ such that } \forall i,j \text{ we have } |\text{CallSigns}(f_i^j)| = 1
\]
converted into QBF instances (instead of DQBF), we employ multiple-function-multiple-callsign, multiple-function-single-callsign, BV-constrained synthesis and syntax-guided synthesis with different synthesis strategies: DQBF based synthesis, as CVC4\textit{cex} performs the other state-of-the-art SyGuS tools significantly. Since the single-function-single-callsign instances can be converted into QBF instances (instead of DQBF), we employ the state-of-the-art QBF solvers CADET [Rabe, 2019] and DepQBF [Lonsing and Egly, 2017], Manthan [Golia et al., 2020] for these instances.

With respect to RQ1, it turns-out that CVC4 performs better with BV-constrained synthesis than with syntax-guided synthesis, as CVC4\textit{cex} could solve 602 instances whereas CVC4\textit{enum} could solve only 488. However, CVC4\textit{enum} outperforms the other state-of-the-art SyGuS tools significantly. The second best SyGuS tool was EUSolver which could solve only 236 instances\(^3\).

Table 3 represents the overall analysis for all four categories of SyGuS instances. As shown in Table 3, DQBF solvers and CVC4\textit{cex} have similar performance in terms of number of instances solved. Therefore, concerning RQ2, DQBF solvers perform on par to domain specific synthesis tools, in-fact they perform slightly better for single-invocation-single-callsign category as CADET could synthesize a function for 3 more instances than CVC4\textit{cex}.

### Performance Analysis for BV-constrained synthesis of QBF

We used SyGuS instances to evaluate the performance of different synthesis strategies: DQBF based synthesis, BV-constrained synthesis and syntax-guided synthesis. We further divide the SyGuS instances into four sub-categories: single-function-single-callsign, single-function-multiple-callsign, multiple-function-single-callsign, multiple-function-multiple-callsign.

Since the single-function-single-callsign instances can be converted into QBF instances (instead of DQBF), we employ the state-of-the-art QBF solvers CADET [Rabe, 2019] and DepQBF [Lonsing and Egly, 2017], Manthan [Golia et al., 2020] for these instances.

With respect to RQ1, it turns-out that CVC4\textit{cex} performs better with BV-constrained synthesis than with syntax-guided synthesis, as CVC4\textit{cex} could solve 602 instances whereas CVC4\textit{enum} could solve only 488. However, CVC4\textit{enum} outperforms the other state-of-the-art SyGuS tools significantly. The second best SyGuS tool was EUSolver which could solve only 236 instances\(^3\).

Table 3 represents the overall analysis for all four categories of SyGuS instances. As shown in Table 3, DQBF solvers and CVC4\textit{cex} have similar performance in terms of number of instances solved. Therefore, concerning RQ2, DQBF solvers perform on par to domain specific synthesis tools, in-fact they perform slightly better for single-invocation-single-callsign category as CADET could synthesize a function for 3 more instances than CVC4\textit{cex}.

### Performance Analysis for QBF Instances

We performed an experiment with the SyGuS language as a representation language instead of BVF. We considered QBF instances instead of DQBF as majority of our synthesis benchmarks could reduced to QBF, and QBF is a generalization of DQBF. As CVC4\textit{cex} performed the best amongst the different tools for BV-constrained synthesis for SyGuS instances, we considered the CVC4\textit{cex} to evaluate the performance over 609 SyGuS language representation of QBF instances. Table 4 represents the performance analysis. We performed experiments with two settings: the same timeout (900s) as used for the tools in Section 5.3, and a more relaxed timeout of 7200s. With the 900s timeout, CVC4\textit{cex} could solve only 2 instances out of 609 total instances, whereas, Manthan preformed the best by synthesizing the functions for 276 instances. With the 7200s timeout, CVC4\textit{cex} could not solve any new instances while Manthan solved another 80 instances. Hence, BV-constrained synthesis is not an efficient approach for DQBF solving, which answers RQ3.

### 6 Conclusion

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. It is important to acknowledge that not every application of SyGuS employs grammar as a tool to aid solver efficiency; grammar can also be used as a specification tool such as to ensure that the synthesized program does not leak information [Ardeshiricham et al., 2019]. Since properties such as information flow leakage can also be expressed as hyper-properties, we hope that our results will motivate the study of synthesis formulation with richer specifications.

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.

It is certainly worth remarking that our experimental results were presented for the case when $\mathbb{T}$ was restricted to BV, and therefore, a valid criticism to the above proposed hypothesis would be lack of evidence for theories beyond bitvectors. In this regard, an interesting direction of future research would be design of techniques for DQF($\mathbb{T}$).

### Acknowledgments

This work was supported in part by National Research Foundation Singapore under its NRF Fellowship Programme [NRF-NRFFA11-2019-0004 ] and AI Singapore Programme [AISG-RP-2018-005], and NUS ODPRT Grant [R-252-000-685-13]. The computational work for this article was performed on resources of the National Supercomputing Centre, Singapore: https://www.nscc.sg.

---

\(^3\)Detailed discussion is deferred to technical report https://arxiv.org/abs/2105.09221
References


