Certified DQBF Solving by Definition Extraction

Franz-Xaver Reichl, Friedrich Slivovsky(✉), and Stefan Szeider

TU Wien, Vienna, Austria
{freichl,fs,sz}@ac.tuwien.ac.at

Abstract. We propose a new decision procedure for dependency quantified Boolean formulas (DQBFs) that uses interpolation-based definition extraction to compute Skolem functions in a counter-example guided inductive synthesis (CEGIS) loop. In each iteration, a family of candidate Skolem functions is tested for correctness using a SAT solver, which either determines that a model has been found, or returns an assignment of the universal variables as a counterexample. Fixing a counterexample generally involves changing candidates of multiple existential variables with incomparable dependency sets. Our procedure introduces auxiliary variables—which we call *arbiter variables*—that each represent the value of an existential variable for a particular assignment of its dependency set. Possible repairs are expressed as clauses on these variables, and a SAT solver is invoked to find an assignment that deals with all previously seen counterexamples. Arbiter variables define the values of Skolem functions for assignments where they were previously undefined, and may lead to the detection of further Skolem functions by definition extraction.

A key feature of the proposed procedure is that it is certifying by design: for true DQBF, models can be returned at minimal overhead. Towards certification of false formulas, we prove that clauses can be derived in an expansion-based proof system for DQBF.

In an experimental evaluation on standard benchmark sets, a prototype implementation was able to match (and in some cases, surpass) the performance of state-of-the-art-solvers. Moreover, models could be extracted and validated for all true instances that were solved.

1 Introduction

Sustained progress in propositional satisfiability (SAT) solving [23] has resulted in a growing number of applications in the area of electronic design automation [49], such as model checking [7], synthesis [43], and symbolic execution [5]. Efficient SAT solvers were essential for recent progress in constrained sampling and counting [31], two problems with many applications in artificial intelligence. In these cases, SAT solvers are used to deal with problems from complexity classes beyond NP and propositional encodings that grow super-polynomially in the size of the original instances. As a consequence, these problems are not directly encoded in propositional logic but have to be reduced to a sequence of SAT instances.

C.-M. Li and F. Manyà (Eds.): SAT 2021, LNCS 12831, pp. 499–517, 2021.
https://doi.org/10.1007/978-3-030-80223-3_34

The success of SAT solving on the one hand, and the inability of propositional logic to succinctly encode problems of interest on the other hand, have prompted the development of decision procedures for more succinct generalizations of propositional logic such as Quantified Boolean Formulas (QBFs). Evaluating QBFs is PSPACE-complete [45] and thus believed to be much harder than SAT, but in practice the benefits of a smaller encoding may outweigh the disadvantage of slower decision procedures [13]. A QBF is true if it has a *model*, which is a family of Boolean functions (often called *Skolem functions*) that satisfy the matrix of the input formula for each assignment of universal variables. The arguments of each Skolem function are implicitly determined by the nesting of existential and universal quantifiers. Dependency QBF (DQBF) explicitly state a *dependency set* for each existential variable, which is a subset of universal variables allowed as arguments of the corresponding Skolem function [3,4]. As such, they can succinctly encode the existence of Boolean functions subject to a set of constraints [34], and problems like equivalence checking of partial circuit designs [19] and bounded synthesis [13] can be naturally expressed in this way.

Several decision procedures for DQBF have been developed in recent years (see Sect. 5). Conceptually, these solvers either reduce to SAT or QBF by instantiating [16] or eliminating universal variables [18,20,39,50], or lift Conflict-Driven Clause Learning (CDCL) to non-linear quantifier prefixes by imposing additional constraints [15,47].[1] We believe these methods should be complemented with algorithms that directly reason at the level of Skolem functions [35]. A strong argument in favor of such an approach is the fact that DQBF instances often have a large fraction of unique Skolem functions that can be obtained by definition extraction, but the current solving paradigms have no direct way of exploiting this [40].

In this paper, we develop new decision procedures for DQBF designed around computing Skolem functions by definition extraction. We first describe a simple algorithm that proceeds in two phases. In the first phase, it introduces clauses to make sure each existential variable is defined in terms of its dependency set and auxiliary *arbiter variables*. In the second phase, it searches for an assignment of the arbiter variables under which the definitions are a model. Runs of this algorithm can degenerate into an exhaustive instantiation of dependency sets for easy cases, so we propose an improved version in the Counter-Example Guided Inductive Synthesis (CEGIS) paradigm [28,42,43].

We implemented the CEGIS algorithm in a system named Pedant. In an experimental evaluation, Pedant performs very well compared to a selection of state-of-the-art solvers—notably, it achieves good performance without the aid of the powerful preprocessor HQSPre [51]. One of the benefits of its function-centric design is that Pedant internally computes a family of Skolem functions, and can output models of true instances at a negligible overhead. Using a simple workflow, we are able to validate models for all true instances solved by Pedant. Towards validation of false instances, we prove that clauses introduced by Pedant can be derived in the ∀Exp+Res proof system [6,27].

[1] An approach that does not fit this simplified classification is the First-Order solver iProver [29].

The remainder of the paper is structured as follows. After covering basic concepts in Sect. 2, we present the new decision algorithms for DQBF and prove their correctness in Sect. 3. We describe the implementation and experimental results in Sect. 4. We discuss related work in Sect. 5, before concluding with an outlook on future work in Sect. 6.

2 Preliminaries

Propositional Logic. A *literal* is either a variable or the negation of a variable. A *clause* is the disjunction of literals. A *term* is a conjunction of literals. A formula is in *Conjunctive Normal Form* (CNF) if it is a conjunction of clauses. Whenever convenient, we identify a CNF with a set of clauses and clauses, respectively terms, with sets of literals. We denote the set of variables occurring in a formula φ by $var(\varphi)$. We denote the truth value *true* by TRUE and *false* by FALSE. An *assignment* of a set V of variables is a function mapping V to $\{\text{TRUE}, \text{FALSE}\}$. We denote the set of all assignments for V by $[V]$. Moreover, we associate an assignment σ with the term $\{x \mid x \in \mathbf{dom}(\sigma), \sigma(x) = \text{TRUE}\} \cup \{\neg x \mid x \in \mathbf{dom}(\sigma), \sigma(x) = \text{FALSE}\}$. Whenever convenient, we treat assignments as terms. Let $\sigma \in [V]$ and let $W \subseteq V$, then we denote the *restriction of* σ *to* W by $\sigma|_W$. For a formula φ and an assignment σ we denote the *evaluation of* φ *by* σ with $\varphi[\sigma]$. A formula φ is *satisfied* by an assignment σ if $\varphi[\sigma] = \text{TRUE}$ and it is *falsified* by σ otherwise. A formula φ is *satisfiable* if there is an assignment σ that satisfies φ and it is *unsatisfiable* otherwise. Let φ and ψ be two formulae, φ *entails* ψ, denoted by $\varphi \vDash \psi$, if every assignment satisfying φ also satisfies ψ. A *definition* for a variable x by a set of variables X in a formula φ is a formula ψ with $var(\psi) \subseteq X$ such that for every satisfying assignment σ of φ the equality $\sigma(x) = \psi[\sigma]$ holds [40].

Dependency Quantified Boolean Formulas. We only consider *Dependency Quantified Boolean formulas* (DQBF) in *Prenex Conjunctive Normal Form* (PCNF). A DQBF in PCNF is denoted by $\Phi = \mathcal{Q}.\varphi$, where *(a)* the *quantifier prefix* $\mathcal{Q}$ is given by $\mathcal{Q} = \forall u_1 \ldots \forall u_n \exists e_1(D_1) \ldots \exists e_m(D_m)$. Here $u_1, \ldots, u_n$ and $e_1, \ldots, e_m$ shall be pairwise different variables. We denote the set $\{u_1, \ldots, u_n\}$ by U_Φ and the set $\{e_1, \ldots, e_m\}$ by E_Φ. Additionally, $D_1, \ldots, D_m$ shall be subsets of U_Φ. *(b)* the *matrix* φ shall be a CNF with $var(\varphi) \subseteq U_\Phi \cup E_\Phi$. For $1 \leq i \leq m$ we call the set D_i the *dependencies* of e_i. We refer to the variables in U_Φ as *universal variables* and to the variables in E_Φ as *existential variables*. For an existential variable e we denote its dependencies by $D_\Phi(e)$. If the underlying DQBF is clear from the context we omit the subscript.

Let Φ be a DQBF and F be a set of functions $\{f_{e_1}, \ldots, f_{e_m}\}$ such that for $1 \leq i \leq m$, $f_{e_i} : [D_i] \to \{\text{TRUE}, \text{FALSE}\}$. For an assignment σ to the universal variables we denote the existential assignment $\{f_{e_1}(\sigma|_{D_1}), \ldots, f_{e_m}(\sigma|_{D_m})\}$ by $F(\sigma)$. F is a *model* (or a *winning* $\exists$*-strategy*) for Φ if for each assignment σ to the universal variables, the assignment $\sigma \cup F(\sigma)$ satisfies the matrix φ. A DQBF is true if it has a model and false otherwise.

$\forall Exp{+}Res$. The *DQBF-*$\forall Exp{+}Res$ [6] calculus is a proof system for DQBF, which is based on the $\forall Exp{+}Res$ calculus for QBF. It instantiates the matrix of a DQBF with a universal assignment and uses propositional resolution on the instantiated clauses. This proof system is sound and refutationally complete [6]. Since we are interested in DQBF, we refer to DQBF-∀Exp+Res simply as ∀Exp+Res.

3 Solving DQBF by Definition Extraction

In this section, we describe two decision procedures for DQBF that leverage definition extraction. We start with an algorithm (Algorithm 1) that is fairly simple but introduces some important concepts. Because this algorithm leads to the equivalent of exhaustive expansion of universal variables on trivial examples, we then introduce a more sophisticated algorithm based on CEGIS (Algorithm 2). We also sketch correctness proofs for both algorithms.

Throughout this section, we consider a fixed DQBF $\Phi := \mathcal{Q}.\ \varphi$ with quantifier prefix $\mathcal{Q} := \forall u_1 \ldots \forall u_n \exists e_1(D_1) \ldots \exists e_m(D_m)$.

3.1 A Two-Phase Algorithm

The algorithm proceeds in two phases. In the first phase (GenerateDefinitions), it finds definitions ψ_{Def} for all existential variables. It maintains a set A of auxiliary *arbiter variables* whose semantics are encoded in a set φ_A of *arbiter clauses*, both of which are empty initially. If a variable e_i is defined in terms of its dependency set, the definition is computed using a SAT solver (line 15) capable of generating interpolants [40]. Otherwise, the SAT solver returns an assignment ξ of the dependency set D_i and the arbiter variables A for which the variable is not defined. In particular, e_i is not defined under the restriction $\sigma = \xi|_{D_i}$ to its dependency set. The algorithm then introduces an arbiter variable e_i^σ that determines the value of the Skolem function for e_i under σ. In subsequent iterations, we include these arbiter variables in the set of variables that can be used in a definition of e_i. The newly introduced clauses ensure that e_i and e_i^σ take the same value under the assignment σ (line 13), so that e_i is defined by e_i^σ and D_i. Since the number of assignments σ of the dependency set is bounded, we will eventually find a definition of e_i in terms of its dependency set D_i and the arbiter variables A.

In the second phase (FindArbiterAssignment), a SAT solver (line 21) is used to find an assignment of the arbiter variables under which the definitions obtained in the first phase are a model. Starting with an initial assignment τ, we use a SAT solver to check whether the formula $\psi_{Def} \wedge \neg\varphi$ consisting of the definitions from the first phase and the negated matrix of the input DQBF is unsatisfiable under τ (line 23). If that is the case, Algorithm 1 returns TRUE. Otherwise, the SAT solver returns an assignment σ as a counterexample. Since the existential variables are defined in $\varphi \wedge \varphi_A$ by the universal and arbiter variables, the formula $\varphi \wedge \varphi_A$ must be unsatisfiable under the assignment $\tau \wedge \sigma|_U$ consisting of the arbiter assignment and counterexample restricted to universal

variables. A *core* ρ of failed assumptions $\tau \wedge \sigma_U$ such that $\rho \models \neg(\varphi \wedge \varphi_A)$ is extracted using another SAT call. The assignment $\rho|_A$ represents a concise reason for the failure of the arbiter assignment τ, and its negation $\neg\rho|_A$ is added as a new clause to the SAT solver used to generate arbiter assignments, which is subsequently invoked to find a new arbiter assignment.

This process continues until a model is found or the SAT solver cannot find a new arbiter assignment, in which case the algorithm returns FALSE.

Algorithm 1. Solving DQBF by Definition Extraction

procedure SolveByDefinitionExtraction(Φ)
  $(\varphi_A, A, \psi_{Def}) \leftarrow$ GenerateDefinitions(Φ)
  return FindArbiterAssignment($\Phi, \varphi_A, A, \psi_{Def}$)

procedure GenerateDefinitions(Φ)
  $\triangleright\ \Phi = \forall u_1 \ldots \forall u_n \exists e_1(D_1) \ldots \exists e_m(D_m).\varphi$
  $A \leftarrow \emptyset,\ \psi_{Def} \leftarrow \emptyset,\ \varphi_A \leftarrow \emptyset$
  $\triangleright$ A: arbiter variables, ψ_{Def}: definitions, φ_A: arbiter clauses
  for $i = 1, \ldots, m$ **do**
    isDefined, $\xi \leftarrow$ isDefined($e_i, A \cup D_i, \varphi \wedge \varphi_A$)
    while not *isDefined* **do**
      $\sigma \leftarrow \xi|_{D_i}$ $\triangleright$ e_i is not defined under $\xi \in [D_i \cup A]$
      $A \leftarrow A \cup \{e_i^\sigma\}$
      $\varphi_A \leftarrow \varphi_A \wedge (e_i^\sigma \vee \neg\sigma \vee \neg e_i) \wedge (\neg e_i^\sigma \vee \neg\sigma \vee e_i)$
      isDefined, $\xi \leftarrow$ isDefined($e_i, A \cup D_i, \varphi \wedge \varphi_A$)
    $\psi_{Def}^i \leftarrow$ getDefinition($e_i, A \cup D_i, \varphi \wedge \varphi_A$)
    $\psi_{Def} \leftarrow \psi_{Def} \wedge (e_i \leftrightarrow \psi_{Def}^i)$
  return $(\varphi_A, A, \psi_{Def})$

procedure FindArbiterAssignment($\Phi, \varphi_A, A, \psi_{Def}$)
  $\tau \leftarrow \bigwedge_{a \in A} a$ $\triangleright$ initial assignment to the arbiter variables
  validitySolver $\leftarrow$ SatSolver($\psi_{Def} \wedge \neg\varphi$)
  arbiterSolver $\leftarrow$ SatSolver($\emptyset$)
  loop
    if *validitySolver*.solve(τ) **then**
      $\sigma \leftarrow$ *validitySolver*.getModel()
      $\rho \leftarrow$ getCore($\varphi \wedge \varphi_A, \tau \wedge \sigma|_U$)
      arbiterSolver.addClause($\neg\rho|_A$)
      if *arbiterSolver*.solve() **then**
        $\tau \leftarrow$ *arbiterSolver*.getModel()
      else
        return FALSE
    else
      return TRUE

We now argue that Algorithm 1 is a decision procedure for DQBF. Due to space constraints, some proofs are omitted. In the following, A shall denote a set of arbiter variables and φ_A shall denote the associated set of arbiter clauses.

A DQBF has a model if, and only if, there is a propositional formula for each existential variable that defines its Skolem functions using only variables from the dependency set. This can be slightly generalized by allowing the definition to contain existential variables whose dependency sets are a subset.

Lemma 1. *Let Φ be a DQBF and $<_E$ a linear ordering of its existential variables. Then Φ is true if, and only if, for each $e \in E$ there is a formula ψ_e with $var(\psi_e) \subseteq D(e) \cup \{x \in E \mid D(x) \subseteq D(e), x <_E e\}$ such that $\neg\varphi \wedge \bigwedge_{e \in E}(e \leftrightarrow \psi_e)$ is unsatisfiable.*

Theorem 1. *If Algorithm 1 returns* TRUE *for the DQBF Φ then Φ is true.*

Proof. Let $\Phi' := \mathcal{Q}\exists A(\emptyset).\varphi$, and let $<_E$ be any ordering of existential variables in Φ' in which the variables in A come before the remaining variables. If Algorithm 1 returns TRUE, we know that there is an arbiter assignment τ such that $\neg\varphi \wedge \psi_{Def} \wedge \tau$ is unsatisfiable. For each arbiter variable e^σ, we obtain a definition ψ_e^σ as $\psi_e^\sigma := \tau(e^\sigma)$. We can now replace the arbiter assignment τ with these definitions and apply Lemma 1 to conclude that Φ' is true. But if Φ' is true, then necessarily also Φ is true. □

To show that the algorithm returns FALSE only if the input DQBF is false, one can prove that clauses on arbiter variables introduced by FINDARBITERASSIGNMENT can be derived (as clauses on annotated literals) in ∀Exp+Res.

Proposition 1. *For each clause C added to the arbiter solver by Algorithm 1 (line 26), a clause $C' \subseteq C$ can be derived from Φ in ∀Exp+Res.*

Theorem 2. *If Algorithm 1 returns* FALSE *for the DQBF Φ then Φ is false.*

Proof. If the algorithm returns FALSE then the set $\mathcal{C}$ of clauses in the arbiter solver is unsatisfiable. By Proposition 1 for each $C \in \mathcal{C}$ we can derive a clause $C' \subseteq C$ subsuming C in ∀Exp+Res, so there is a ∀Exp+Res refutation of Φ. As ∀Exp+Res is sound [6], this shows that Φ is false. □

Finally, Algorithm 1 terminates since at most one arbiter variable is introduced for each existential variable and assignment of its dependency set in the first phase, and there is a limited number of clauses on arbiter variables that can be introduced in the second phase. In combination with Theorem 1 and Theorem 2, we obtain the following result.

Corollary 1. *Algorithm 1 is a decision procedure for DQBF.*

3.2 Combining Definition Extraction with CEGIS

Discounting SAT calls, the running time of Algorithm 1 is essentially determined by the number of assignments of a dependency set for which the corresponding existential variable is not defined: it introduces an arbiter variable for each such

assignment in the first phase, and the number of iterations in the second phase is bounded by the number of arbiter assignments. As a result, even a single existential variable that is unconstrained and has a large dependency set causes the algorithm to get stuck enumerating universal assignments.

A key insight underlying the success of counter-example guided solvers for QBF [25,26,46] is that it is typically overkill to perform complete expansion of universal variables. Instead, they incrementally refine Skolem functions by taking into account universal assignments that pose a problem for the current solution candidate.[2]

Following this idea, we now present an improved algorithm (Algorithm 2) in the style of Counter-Example Guided Inductive Synthesis (CEGIS) [28]. It integrates the two phases of Algorithm 1 into a single loop. In each iteration, it first tries to find definitions for existential variables in terms of their dependency sets and the arbiter variables (FindDefinitions). The algorithm then proceeds to a validity check of the definitions under the current arbiter assignment (CheckArbiterAssignment). A key difference to Algorithm 1 is that we may not have a definition for each variable at this point. In this case, we can simply leave the existential variable unconstrained in the SAT call except for arbiter clauses φ_A (and *forcing clauses* φ_F, which we discuss later). In the implementation, we limit the SAT solver's freedom to generate counterexamples by substituting a default value or a heuristically obtained "guess" for the Skolem function. Here, any function on variables from the dependency set can be used without affecting correctness, one only has to make sure that counterexamples are not repeated to guarantee termination.

If a counterexample σ is found, procedure CheckArbiterAssignment returns it to the main loop. Otherwise, (line 25), we have to check whether the SAT call in line 20 returned UNSAT because a model has been found, or whether there is an inconsistency in the formula $\varphi_A \wedge \varphi_F$ comprised of arbiter and forcing clauses under the current arbiter assignment τ. The procedure CheckConsistency either finds that the model is consistent, in which case Algorithm 2 returns TRUE, or else computes an assignment σ of the universal variables as a counterexample. If CheckArbiterAssignment returns FALSE, the main loop resumes in line 14 with a call to AnalyzeConflict.

To see what this procedure does, let us first consider the simple case in which the counterexample σ only contains an assignment of universal variables that was returned by the consistency check. Then, the existential assignment $\rho_\exists = \emptyset$ is empty, the for-loop is skipped and no new arbiter variables are introduced (line 55), and the procedure only tries to further simplify the failed arbiter assignment ρ_A in line 58, before adding its negation to the arbiter solver.

Now assume $\rho_\exists$ is nonempty but the case distinction in the body of the for-loop between lines 43 and 54 always leads to line 51. Then $notforced = \rho_\exists$ and the procedure NewArbiters creates new arbiter variables A' and clauses φ'_A for each existential variable $e \in \mathbf{dom}(\rho_\exists)$ and the universal counterexample $\sigma_\forall$

[2] In these QBF solvers, Skolem functions are typically only indirectly represented by trees of formulas (*abstractions*) that encode viable assignments.

(restricted to the dependency set $D(e)$ in each case). Since these arbiter variables determine the assignment of the existential variables in $\mathbf{dom}(\rho_\exists)$ under $\sigma_\forall$, we can replace $\rho_\exists$ with the assignment $\rho'_A := \{e^\xi \in A' \mid e \in \rho_\exists\} \cup \{\neg e^\xi \in A' \mid \neg e \in \rho_\exists\}$ (line 57) and conclude that $\varphi \wedge \varphi_A \wedge \varphi_F$ is unsatisfiable under the assignment $\rho_A \cup \rho'_A \cup \sigma_\forall$, which only assigns arbiter variables and universal variables. A clause forbidding the arbiter assignment $\rho_A \cup \rho'_A$ can now be added as before.

Finally, let us turn to the general case, which includes entailment checks for each existential literal $\ell \in \rho_\exists$ in the minimized counterexample. These checks are added to reduce the number of new arbiter variables created. If the literal ℓ is entailed by the assignment $\sigma_\forall \wedge \tau$, we add further literals from τ to the failed arbiter assignment ρ_A (if necessary) to ensure that ℓ is entailed by $\sigma_\forall \wedge \rho_A$. No arbiter variable has to be introduced for $var(\ell)$ in this case. Otherwise, if $\neg\ell$ is entailed by $\sigma_\forall \wedge \tau$, then the counterexample is spurious since $e = var(\ell)$ must be assigned the opposite way under $\sigma_\forall \wedge \tau$ by any Skolem function. To enforce this in the next iteration, the algorithm adds a *forcing clause* C encoding the implication $\sigma_\forall \wedge \tau \rightarrow \neg\ell$ (which can be further strengthened by restricting $\sigma_\forall$ to the dependency set of e) to φ_F. It also sets a flag *oppositeForced*, which causes AnalyzeConflict to exit with true instead of adding new arbiter variables.

If AnalyzeConflict returns true, Algorithm 2 proceeds to the next iteration of its main loop with the same arbiter assignment τ but additional forcing clauses. Otherwise, AnalyzeConflict returns false after adding a clause to the SAT solver *arbiterSolver*, and FindNewArbiterAssignment is called to determine a new arbiter assignment τ that satisfies all previously added clauses. Algorithm 2 terminates either when it discovers a model or when it cannot find a new arbiter assignment.

We now sketch a proof that shows that Algorithm 2 is a decision procedure for DQBF. As in Sect. 3.1, A denotes a set of arbiter variables and φ_A denotes the associated set of arbiter clauses.

Definition 1 (Forcing Clause). *Let ℓ be an existential literal, ψ a formula with $var(\psi) \subseteq U \cup E \cup A$ and let σ be a (partial) assignment for $U \cup A$. We say that ℓ is* forced *by σ in ψ if $\psi \wedge \sigma \wedge \neg\ell$ is unsatisfiable. If ℓ is forced by σ then $\neg\sigma|_{D(var(\ell))\cup A} \vee l$ is a* forcing clause.

In particular, if a literal ℓ is forced by an assignment σ in a formula φ then $\varphi \wedge \sigma \vDash \ell$ holds. Forcing clauses can be added to a DQBF without changing its models. In particular, the resulting DQBF has the same truth value.

Lemma 2. *Let $C_1, \ldots, C_k$ be clauses such that for each index i, the clause C_i is a forcing clause in $\varphi \wedge \varphi_A \wedge \bigwedge_{1\leq j<i} C_j$. Then the DQBF $\mathcal{Q}\exists A(\emptyset).\varphi \wedge \varphi_A \wedge \bigwedge_{1\leq i\leq k} C_i$ is true if and only if Φ is true.*

Theorem 3. *If Algorithm 2 returns* true *for a DQBF Φ then Φ is true.*

Proof. We assume that the algorithm returns true and show that the DQBF Φ is true. We know that we have a set A of arbiter variables, a set φ_A of arbiter clauses, a set φ_F of forcing clauses and a set $E' \subseteq E$ such that each $e \in E'$ has

Algorithm 2. Solving DQBF by Definition Extraction (CEGIS Version)

```
1: procedure SolveByDefinitionExtractionCEGIS(Φ)
2:     ▷ Φ = ∀u_1 ... ∀u_n ∃e_1(D_1) ... ∃e_m(D_m).φ
3:     ▷ A: arbiter variables, ψ_Def: definitions, φ_A: arbiter clauses
4:     A ← ∅, ψ_Def ← ∅, φ_A ← ∅
5:     φ_F ← ∅                                          ▷ forcing clauses
6:     τ ← ∅                                            ▷ arbiter assignment
7:     arbiterSolver ← SatSolver(∅)
8:     loop
9:         ψ_Def ← FindDefinitions({e ∈ E | e undefined}, φ ∧ φ_A ∧ φ_F)
10:        modelValid, σ ← CheckArbiterAssignment(τ)
11:        if modelValid then
12:            return true
13:        ▷ σ is a counterexample
14:        if AnalyzeConflict(σ) then
15:            ▷ forcing clauses have been added to φ_F
16:            continue
17:        if not FindNewArbiterAssignment() then
18:            return false

19: procedure CheckArbiterAssignment(τ)
20:     checker ← SatSolver(¬φ ∧ ψ_Def ∧ φ_F ∧ φ_A)
21:     if checker.solve(τ) then
22:         σ ← checker.values(E ∪ U)
23:         return false, σ
24:     else
25:         isConsistent, σ ← CheckConsistency(φ_A ∧ φ_F, τ)
26:         if isConsistent then
27:             return true, ∅
28:         else
29:             ▷ σ ∈ [U] is such that φ_A ∧ φ_F ∧ τ ∧ σ is unsatisfiable
30:             return false, σ

31: procedure FindNewArbiterAssignment()
32:     if arbiterSolver.Solve() then
33:         τ ← arbiterSolver.getModel()|_A
34:         return true
35:     return false
```

a definition ψ_e in $\varphi \wedge \varphi_A \wedge \varphi_F$ by $D(e) \cup A$. Let $\psi_{Def} := \bigwedge_{e \in E'}(e \leftrightarrow \psi_e)$ and $\Phi' := \mathcal{Q}\exists A(\emptyset).\varphi \wedge \varphi_A \wedge \varphi_F$. By Lemma 2, we know that Φ is true if and only if Φ' is true. As the algorithm returns TRUE, we know that $\neg\varphi \wedge \varphi_A \wedge \varphi_F \wedge \psi_{Def}$ is unsatisfiable. Using a mild generalization of Lemma 1, this implies that Φ' is true. Thus, Φ is true as well. □

As in the case of Algorithm 1, the correctness of FALSE answers for Algorithm 2 follows from a correspondence with ∀Exp+Res derivations.

```
36: procedure AnalyzeConflict(σ)
37:     σ∀ ← σ|U                                  ▷ σ∀ assigns all universal variables
38:     ρ ← getCore(φ ∧ φA ∧ φF, σ ∧ τ)
39:     ρ∃ ← ρ|E, ρA ← ρ|A
40:     notForced ← ∅                             ▷ collect literals ℓ ∈ ρ∃ that are not implied
41:     oppositeForced ← false
42:     ψ ← φ ∧ φA ∧ φF
43:     for ℓ ∈ ρ∃ do
44:         if ψ ∧ σ∀ ∧ τ ⊨ ℓ then
45:             ρ ← getCore(ψ, σ∀ ∧ τ ∧ ¬ℓ)
46:             ρA ← ρA ∪ ρ|A                     ▷ add reason for ℓ to failed arbiter assignment ρA
47:         else if ψ ∧ σ∀ ∧ τ ⊨ ¬ℓ then
48:             φF ← φF ∧ getForcingClause(ψ, σ∀ ∧ τ, ¬ℓ)
49:             oppositeForced ← true
50:         else
51:             notForced ← notForced ∪ {ℓ}
52:     if oppositeForced then
53:         return true
54:     ▷ no literal was forced to the opposite polarity
55:     φ'A, A' ← newArbiters(notForced, σ∀)
56:     φA ← φA ∧ φ'A
57:     ρA ← ρA ∧ setAssignment(A', ρ∃)
58:     ρA ← getCore(ψ, ρA ∧ σ∀)|A
59:     arbiterSolver.addClause(¬ρA)
60:     return false
```

Proposition 2. *For each clause C added to the arbiter solver by Algorithm 2 (line 59), a clause $C' \subseteq C$ can be derived from Φ in $\forall$Exp+Res.*

Theorem 4. *If Algorithm 2 returns* false *for a DQBF Φ then Φ is false.*

Each iteration of Algorithm 2 introduces new forcing clauses or forbids another arbiter assignment. Because there is a bound on the number of arbiter variables that can be introduced, the number of such clauses can be bounded as well, and the algorithm eventually terminates. Together with Theorem 3 and Theorem 4, this gives rise to the following corollary.

Corollary 2. *Algorithm 2 is a decision procedure for DQBF.*

4 Experiments

We implemented Algorithm 2 as described in the previous section in a prototype named Pedant.[3] For definition extraction, it uses a subroutine from Unique [40] that in turn relies on an interpolating version of MiniSat [12] bundled with the ExtAvy model checker [22,48]. Further, CaDiCaL is used as a

[3] Available at https://github.com/perebor/pedant-solver.

SAT solver [8] (we also tested with CryptoMiniSAT [44] and Glucose [2] but saw no significant differences in overall performance). Pedant can read DQBF in the standard DQDIMACS format and output models in the DIMACS format.

The implementation incorporates a few techniques not explicitly mentioned in the above pseudocode. We identify *unate* existential literals (a generalization of pure literals) [1], which can be used in any model of a DQBF. Moreover, we set a (configurable) default value for existential variables that applies when there is no forcing clause propagating a different value. This is to limit the freedom of the SAT solver used in the validity check in coming up with counterexamples. Moreover, when checking for definability of an existential variable, we use *extended dependencies* that include existential variables with dependency sets that are contained in the dependencies of the variable that is checked.

For all experiments described below we use a cluster with Intel Xeon E5649 processors at 2.53 GHz running 64-bit Linux.

4.1 Performance on Standard Benchmark Sets

We compare Pedant with other DQBF solvers on standard benchmark sets in terms of instances solved within the timeout and their PAR2 score.[4] Specifically, we choose the solvers dCAQE [47], iDQ [16], HQS [20], and the recently introduced DQBDD [39]. Both HQS and DQBDD internally use HQSPre [51] as a preprocessor. For dCAQE and iDQ, we call HQSPre with a time limit of 300 seconds (the time for preprocessing is included in the total running time). By default, Pedant is run without preprocessing.

The results are based on a single run with a time and memory limit of 1800 seconds and 8 GB, respectively, which are enforced using RunSolver [36].[5] We report results for two benchmark sets. The first—which we refer to as the "Compound" set—has been used in recent papers on HQS [18]. It is comprised of instances encoding partial equivalence checking (PEC) [14,16,19,37] and controller synthesis [10], as well as succinct DQBF representations of propositional satisfiability [4]. Results are summarized in Table 1. Pedant solved the most instances overall and for 4 out of 5 families (the "Balabanov" family being the exception), with DQBDD coming in a close second. The performance of Pedant on the PEC instances in the "Finkbeiner" family is particularly encouraging.

Next, we consider the instances from the DQBF track of QBFEVAL'20 [33]. Results are shown in Table 2. Here, Pedant falls behind the other solvers, with the exception of iDQ. In particular, significantly fewer instances from the "Kullmann" and "Tentrup" families are solved.

For the autarky finding benchmarks in the "Kullmann" family [30], we noticed that most dependencies can be removed by preprocessing with the reflexive resolution-path dependency scheme [41,52]. The resulting instances are much

[4] The Penalized Average Runtime (PAR) is the average runtime, with the time for each unsolved instance calculated as a constant multiple of the timeout.

[5] Due to the heavy-tailed runtime distribution of DQBF solvers, run-to-run variance rarely affects the number of solved instances. However, PAR2 scores should be taken with a grain of salt and only used to compare orders of magnitude.

Table 1. Results for the "Compound" benchmark set.

Family(Total)	dCAQE	DQBDD	HQS	iDQ	Pedant
	Sol/PAR2	Sol/PAR2	Sol/PAR2	Sol/PAR2	Sol/PAR2
Balabanov(34)	**21**/$1.5{\cdot}10^{3}$	13/$2.3{\cdot}10^{3}$	19/$1.8{\cdot}10^{3}$	**21**/$1.5{\cdot}10^{3}$	13/$2.3{\cdot}10^{3}$
Biere(1200)	**1200**/$1.6{\cdot}10^{-1}$	1197/$9.0{\cdot}10^{0}$	**1200**/$6.4{\cdot}10^{-2}$	1184/$6.6{\cdot}10^{1}$	**1200**/$1.0{\cdot}10^{-1}$
Bloem(461)	85/$2.9{\cdot}10^{3}$	82/$3.0{\cdot}10^{3}$	82/$3.0{\cdot}10^{3}$	50/$3.2{\cdot}10^{3}$	**98**/$2.9{\cdot}10^{3}$
Finkbeiner(2000)	32/$3.5{\cdot}10^{3}$	1999/$1.1{\cdot}10^{1}$	1799/$3.9{\cdot}10^{2}$	6/$3.6{\cdot}10^{3}$	**2000**/$1.7{\cdot}10^{0}$
Scholl(1116)	568/$1.8{\cdot}10^{3}$	793/$1.1{\cdot}10^{3}$	676/$1.4{\cdot}10^{3}$	345/$2.5{\cdot}10^{3}$	**854**/$8.7{\cdot}10^{2}$
All(4811)	1906/$2.2{\cdot}10^{3}$	4084/$5.5{\cdot}10^{2}$	3776/$7.9{\cdot}10^{2}$	1606/$2.4{\cdot}10^{3}$	**4165**/$4.9{\cdot}10^{2}$

easier to solve for Pedant, and models can still be validated against the original DQBFs. In general, we found that preprocessing with HQSPre can have both positive and negative effects on Pedant. The rightmost columns of Table 2 show results when preprocessing is enabled.[6] Overall, performance is clearly improved, but fewer instances from the "Bloem" and "Scholl" families are solved. In prior work, it was observed that preprocessing can destroy definitions [40], and this appears to be the case here as well.

For the instances from the "Tentrup" family, we discovered that the performance of Pedant is sensitive to which counterexamples are generated by CaDiCaL. With the right sequence of counterexamples, false instances can be refuted quickly, while otherwise the solver is busy introducing arbiter variables for minor variations of previously encountered cases. Curiously, this also appears to be the case for true instances. We believe that the algorithm can be made more robust against such "adversarial" sequences of counterexamples by achieving better generalization (see Sect. 6).

Table 2. Results for the QBFEVAL'20 DQBF benchmark set.

Family(Total)	dCAQE	DQBDD	HQS	iDQ	Pedant	PedantHQ
	Sol/PAR2	Sol/PAR2	Sol/PAR2	Sol/PAR2	Sol/PAR2	Sol/PAR2
Balabanov(34)	**21**/$1.5{\cdot}10^{3}$	13/$2.3{\cdot}10^{3}$	19/$1.8{\cdot}10^{3}$	**21**/$1.5{\cdot}10^{3}$	14/$2.3{\cdot}10^{3}$	13/$2.4{\cdot}10^{3}$
Bloem(90)	31/$2.4{\cdot}10^{3}$	32/$2.3{\cdot}10^{3}$	33/$2.3{\cdot}10^{3}$	14/$3.1{\cdot}10^{3}$	**37**/$2.2{\cdot}10^{3}$	25/$2.7{\cdot}10^{3}$
Kullmann(50)	35/$1.1{\cdot}10^{3}$	**50**/$1.5{\cdot}10^{1}$	41/$6.9{\cdot}10^{2}$	**50**/$3.4{\cdot}10^{0}$	34/$1.3{\cdot}10^{3}$	40/$7.3{\cdot}10^{2}$
Scholl(90)	52/$1.5{\cdot}10^{3}$	78/$4.9{\cdot}10^{2}$	77/$5.3{\cdot}10^{2}$	15/$3.0{\cdot}10^{3}$	**82**/$3.3{\cdot}10^{2}$	65/$1.2{\cdot}10^{3}$
Tentrup(90)	77/$5.5{\cdot}10^{2}$	**84**/$2.8{\cdot}10^{2}$	78/$5.1{\cdot}10^{2}$	17/$2.9{\cdot}10^{3}$	15/$3.0{\cdot}10^{3}$	**84**/$2.9{\cdot}10^{2}$
All(354)	216/$1.4{\cdot}10^{3}$	**257**/$1.0{\cdot}10^{3}$	248/$1.1{\cdot}10^{3}$	117/$2.4{\cdot}10^{3}$	182/$1.8{\cdot}10^{3}$	227/$1.4{\cdot}10^{3}$

[6] With options `--resolution 1 --univ_exp 0 --substitute 0`.

4.2 Distribution of Defined Existential Variables

The main design goal for PEDANT was to create a solver that benefits from unique Skolem functions given by propositional definitions. We thus expect PEDANT to do well on instances where a large proportion of existential variables is defined. Figure 1 shows the distribution of defined existential variables (i.e., unique Skolem functions) as computed by UNIQUE [40]. These definitions are also found by PEDANT without the introduction of arbiter variables. Comparing Table 1 and Table 2 with Fig. 1, we see that PEDANT performed better for instance families with a larger fraction of defined variables. This makes sense: the fewer variables are undefined, the fewer arbiter variables need to be introduced.

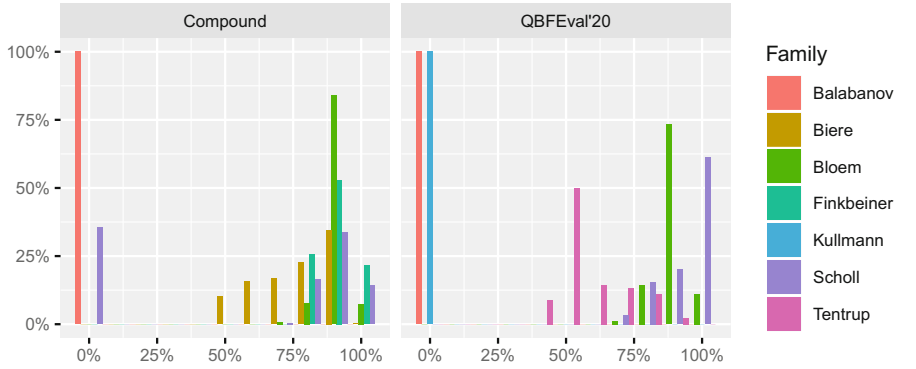

Fig. 1. Distribution of defined variables by benchmark set and family. For a given percentage x_0 on the x-axis, the y-axis shows the fraction of instances from each benchmark family for which x_0 percent of existential variables are defined. For example, the instances in the "Balabanov" family have no defined variables, while the fraction of defined variables for instances in the "Finkbeiner" family ranges from 75% to 100%.

4.3 Solution Validation

When running PEDANT without preprocessing (the default), we had it trace and output models in DIMACS format. We implemented a simple workflow for validating these models in Python 3 using the PYSAT library [24]. First, a simple syntactic check is performed to make sure the encoding of each Skolem function only mentions variables in the dependency set of the corresponding variable. Then, a SAT solver is used to verify that substituting the model ψ for existential variables in the matrix φ of the input DQBF is valid, by testing for each clause $C \in \varphi$ whether $\psi \wedge \neg C$ is unsatisfiable (cf. Lemma 1). In this manner, we are able to validate models for all 648 true DQBFs in the two benchmark sets that were solved by PEDANT without preprocessing. The maximum validation time was 237 s, with a mean of 4.3 s and a median of 0.5 s.

The current validation process is intended as a proof of concept. Since models constructed by Pedant are circuits, we plan to support the AIGER format [9] in the near future, and provide a workflow along the lines of QBFCert [32].

5 Related Work

The DQDPLL algorithm lifts the CDCL algorithm to DQBF [15]. While CDCL solvers are free to assign variables in any order, in DQBF a variable may be assigned only after the variables in its dependency set have been assigned. Moreover, its assignment must not differ between branches in the search tree that agree on the assignment of the dependency set. In DQDPLL, this is enforced by temporary *Skolem clauses* that fix the truth value of a variable for a given assignment of its dependencies. The solver dCAQE lifts clausal abstraction from QBF to DQBF [47]. QBF solvers based on abstraction maintain a propositional formula for each quantifier level that characterizes eligible moves in the evaluation game. These *abstractions* are refined by forbidding moves that are known to result in a loss. Abstractions are linked to each other through auxiliary variables that indicate which clauses are satisfied at different levels. dCAQE organizes variables in a dependency lattice that determines the order in which their abstractions may be solved. This can lead to variables being assigned after variables that do not appear in their dependency sets, and additional consistency checks have to be applied to ensure that Skolem functions do not exploit such spurious dependencies. dCAQE uses *fork resolution* as its underlying proof system [34].

Expansion of universal variables can be successively applied to transform a DQBF into a propositional formula that can be passed to a SAT solver [11]. In practice, the space requirements of fully expanding a DQBF are prohibitive. This can be addressed by only expanding some universal variables, as well as considering only a subset of the clauses generated by expansion. Even though such approaches degenerate into full expansion in the worst case, they can be quite effective. The solver iDQ [16] successively expands a DQBF in a counterexample-guided abstraction refinement (CEGAR) loop. Initially, universal variables in each clause are expanded separately. Satisfiability of the resulting propositional formula is checked by a SAT solver. If it is unsatisfiable, so is the original DQBF. Otherwise, iDQ checks whether any pair of literals with consistent annotations are assigned different truth values in the satisfying assignment. If there are no such literals, a model of the DQBF has been found. Otherwise, clauses containing the corresponding clashing literals are further expanded. The system is inspired by the *Inst-Gen* calculus, the proof system underpinning the First-Order solver iProver [29]. Originally designed for the effectively propositional fragment of first-order logic (EPR), iProver also accepts DQBF as input.

The solver HQS seeks to keep the memory requirements of expansion in check by operating on And-Inverter Graph (AIG) representations of input formulas [20]. It uses expansion alongside several other techniques to transform a DQBF into an equivalent QBF and leverage advances in QBF solving [18,50]. HQS is paired with a powerful preprocessor named HQSPre that provides

an arsenal of additional simplification techniques [51], including an incomplete but efficient method for refuting DQBF by reduction to a QBF encoding [14]. HQSpre is also used in the recently developed solver DQBDD [39], which is similar to HQS but relies on Binary Decision Diagrams (BDDs) instead of AIGs to represent formulas and perform quantifier elimination.

Evaluating DQBF is NEXPTIME complete [3] in general, but some tractable subclasses have been identified in recent work [17,38].

6 Conclusion

We presented a decision algorithm for DQBF that relies on definition extraction to compute Skolem functions inside a CEGIS loop, and evaluated it in terms of the prototype implementation Pedant. While the initial results are very promising, we see significant room for improvement and various directions to pursue in future research. Generally, the approach works well when Skolem functions can be computed by definition extraction for a large fraction of existential variables without introducing too many arbiter variables. During testing, we encountered multiple instances for which conflict analysis was occupied dealing with minor variations of a small number of counterexamples. We believe that this is partly due to arbiter variables being introduced for *complete* assignments of dependency sets. Even if the assignment of some universal variables in the dependency set is irrelevant for a given counterexample, the newly introduced arbiter variables only deal with the counterexample as represented by the complete assignment, and each counterexample obtained by varying the assignment of irrelevant universal variables requires a new set of arbiter variables. To avoid this, we plan to experiment with a variant of the algorithm that introduces arbiter variables for *partial* assignments [16,29].

A different approach to generalizing from counterexamples—one that does not require changes in the underlying proof system—is the use of machine learning. By predicting the pattern common to a sequence of counterexamples, it is possible to deal with it wholesale and avoid an exhaustive enumeration [25]. Moreover, recent work on Boolean Synthesis demonstrates the viability of learning Skolem functions by sampling satisfying assignments [21].

Finally, we plan to explore further applications of interpolation-based definition extraction within our algorithm. Currently, its use is limited to existential variables that are defined by their dependency sets in the input DQBF, or are undefined only in a small number of cases. In addition to that, one could search for "partial" definitions under assignments of the dependency set characterized by formulas, or introduce definitions that are valid under assumptions [35].

Acknowledgements. Supported by the Vienna Science and Technology Fund (WWTF) under the grants ICT19-060 and ICT19-065, and the Austrian Science Fund (FWF) under grant W1255.

References

1. Akshay, S., Chakraborty, S., Goel, S., Kulal, S., Shah, S.: What's hard about boolean functional synthesis? In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10981, pp. 251–269. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96145-3_14
2. Audemard, G., Simon, L.: Predicting learnt clauses quality in modern SAT solvers. In: Boutilier, C. (ed.) IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, pp. 399–404 (2009)
3. Azhar, S., Peterson, G., Reif, J.: Lower bounds for multiplayer non-cooperative games of incomplete information. J. Comput. Math. Appl. **41**, 957–992 (2001)
4. Balabanov, V., Chiang, H.K., Jiang, J.R.: Henkin quantifiers and boolean formulae: a certification perspective of DQBF. Theor. Comput. Sci. **523**, 86–100 (2014)
5. Baldoni, R., Coppa, E., D'Elia, D.C., Demetrescu, C., Finocchi, I.: A survey of symbolic execution techniques. ACM Comput. Surv. **51**(3), 50:1–50:39 (2018)
6. Beyersdorff, O., Blinkhorn, J., Chew, L., Schmidt, R.A., Suda, M.: Reinterpreting dependency schemes: soundness meets incompleteness in DQBF. J. Autom. Reason. **63**(3), 597–623 (2019)
7. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic model checking without BDDs. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-49059-0_14
8. Biere, A., Fazekas, K., Fleury, M., Heisinger, M.: CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In: Balyo, T., Froleyks, N., Heule, M., Iser, M., Järvisalo, M., Suda, M. (eds.) Proceedings of SAT Competition 2020 - Solver and Benchmark Descriptions. Department of Computer Science Report Series B, vol. B-2020-1, pp. 51–53. University of Helsinki (2020)
9. Biere, A., Heljanko, K., Wieringa, S.: AIGER 1.9 and beyond. Tech. Rep. 11/2, Institute for Formal Models and Verification, Johannes Kepler University, Altenbergerstr. 69, 4040 Linz, Austria (2011)
10. Bloem, R., Könighofer, R., Seidl, M.: SAT-based synthesis methods for safety specs. In: McMillan, K.L., Rival, X. (eds.) VMCAI 2014. LNCS, vol. 8318, pp. 1–20. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54013-4_1
11. Bubeck, U., Büning, H.K.: Dependency quantified horn formulas: models and complexity. In: Biere, A., Gomes, C.P. (eds.) SAT 2006. LNCS, vol. 4121, pp. 198–211. Springer, Heidelberg (2006). https://doi.org/10.1007/11814948_21
12. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24605-3_37
13. Faymonville, P., Finkbeiner, B., Rabe, M.N., Tentrup, L.: Encodings of bounded synthesis. In: Legay, A., Margaria, T. (eds.) TACAS 2017. LNCS, vol. 10205, pp. 354–370. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54577-5_20
14. Finkbeiner, B., Tentrup, L.: Fast DQBF refutation. In: Sinz, C., Egly, U. (eds.) SAT 2014. LNCS, vol. 8561, pp. 243–251. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-09284-3_19
15. Fröhlich, A., Kovásznai, G., Biere, A.: A DPLL algorithm for solving DQBF (2012). http://fmv.jku.at/papers/FroehlichKovasznaiBiere-POS12.pdf, presented at Workshop on Pragmatics of SAT (POS)

16. Fröhlich, A., Kovásznai, G., Biere, A., Veith, H.: idq: instantiation-based DQBF solving. In: Berre, D.L. (ed.) POS-14. Fifth Pragmatics of SAT workshop, a workshop of the SAT 2014 conference, part of FLoC 2014 during the Vienna Summer of Logic,Vienna, Austria, 13 July 2014. EPiC Series in Computing, vol. 27, pp. 103–116. EasyChair (2014)
17. Ganian, R., Peitl, T., Slivovsky, F., Szeider, S.: Fixed-parameter tractability of dependency QBF with structural parameters. In: Calvanese, D., Erdem, E., Thielscher, M. (eds.) Proceedings of the 17th International Conference on Principles of Knowledge Representation and Reasoning, KR 2020, pp. 392–402 (2020)
18. Ge-Ernst, A., Scholl, C., Wimmer, R.: Localizing quantifiers for DQBF. In: Barrett, C.W., Yang, J. (eds.) Formal Methods in Computer Aided Design, FMCAD 2019, pp. 184–192. IEEE (2019)
19. Gitina, K., Reimer, S., Sauer, M., Wimmer, R., Scholl, C., Becker, B.: Equivalence checking of partial designs using dependency quantified boolean formulae. In: IEEE 31st International Conference on Computer Design, ICCD 2013, pp. 396–403. IEEE Computer Society (2013)
20. Gitina, K., Wimmer, R., Reimer, S., Sauer, M., Scholl, C., Becker, B.: Solving DQBF through quantifier elimination. In: Nebel, W., Atienza, D. (eds.) Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition, DATE 2015, pp. 1617–1622. ACM (2015)
21. Golia, P., Roy, S., Meel, K.S.: Manthan: a data-driven approach for boolean function synthesis. In: Lahiri, S.K., Wang, C. (eds.) CAV 2020. LNCS, vol. 12225, pp. 611–633. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-53291-8_31
22. Gurfinkel, A., Vizel, Y.: Druping for interpolates. In: FMCAD 2014, pp. 99–106. IEEE (2014)
23. Heule, M.J.H., Järvisalo, M., Suda, M.: SAT competition 2018. J. Satisf. Boolean Model. Comput. **11**(1), 133–154 (2019)
24. Ignatiev, A., Morgado, A., Marques-Silva, J.: PySAT: a python toolkit for prototyping with SAT oracles. In: SAT, pp. 428–437 (2018)
25. Janota, M.: Towards generalization in QBF solving via machine learning. In: McIlraith, S.A., Weinberger, K.Q. (eds.) Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), pp. 6607–6614. AAAI Press (2018)
26. Janota, M., Klieber, W., Marques-Silva, J., Clarke, E.M.: Solving QBF with counterexample guided refinement. Artif. Intell. **234**, 1–25 (2016)
27. Janota, M., Marques-Silva, J.: On propositional QBF expansions and Q-resolution. In: Järvisalo, M., Van Gelder, A. (eds.) SAT 2013. LNCS, vol. 7962, pp. 67–82. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39071-5_7
28. Jha, S., Seshia, S.A.: A theory of formal synthesis via inductive learning. Acta Informatica **54**(7), 693–726 (2017)
29. Korovin, K.: iProver – an instantiation-based theorem prover for first-order logic (system description). In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 292–298. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-71070-7_24
30. Kullmann, O., Shukla, A.: Autarkies for DQCNF. In: Barrett, C.W., Yang, J. (eds.) 2019 Formal Methods in Computer Aided Design, FMCAD 2019, pp. 179–183. IEEE (2019)
31. Meel, K.S., et al.: Constrained sampling and counting: Universal hashing meets SAT solving. In: Darwiche, A. (ed.) Beyond NP, Papers from the 2016 AAAI Workshop. AAAI Workshops, vol. WS-16-05. AAAI Press (2016)

32. Niemetz, A., Preiner, M., Lonsing, F., Seidl, M., Biere, A.: Resolution-based certificate extraction for QBF. In: Cimatti, A., Sebastiani, R. (eds.) SAT 2012. LNCS, vol. 7317, pp. 430–435. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31612-8_33
33. Pulina, L., Seidl, M.: The 2016 and 2017 QBF solvers evaluations (qbfeval'16 and qbfeval'17). Artif. Intell. **274**, 224–248 (2019)
34. Rabe, M.N.: A resolution-style proof system for DQBF. In: Gaspers, S., Walsh, T. (eds.) SAT 2017. LNCS, vol. 10491, pp. 314–325. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66263-3_20
35. Rabe, M.N., Seshia, S.A.: Incremental determinization. In: Creignou, N., Le Berre, D. (eds.) SAT 2016. LNCS, vol. 9710, pp. 375–392. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40970-2_23
36. Roussel, O.: Controlling a solver execution with the runsolver tool. J. Satisf. Boolean Model. Comput. **7**(4), 139–144 (2011)
37. Scholl, C., Becker, B.: Checking equivalence for partial implementations. In: Proceedings of the 38th Design Automation Conference, DAC 2001, pp. 238–243. ACM (2001)
38. Scholl, C., Jiang, J.R., Wimmer, R., Ge-Ernst, A.: A PSPACE subclass of dependency quantified boolean formulas and its effective solving. In: The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, pp. 1584–1591. AAAI Press (2019)
39. Síč, J.: Satisfiability of DQBF using binary decision diagrams. Master's thesis, Masaryk University, Brno, Czech Republic (2020)
40. Slivovsky, F.: Interpolation-based semantic gate extraction and its applications to QBF preprocessing. In: Lahiri, S.K., Wang, C. (eds.) CAV 2020. LNCS, vol. 12224, pp. 508–528. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-53288-8_24
41. Slivovsky, F., Szeider, S.: Soundness of Q-resolution with dependency schemes. Theor. Comput. Sci. **612**, 83–101 (2016)
42. Solar-Lezama, A., Jones, C.G., Bodík, R.: Sketching concurrent data structures. In: Gupta, R., Amarasinghe, S.P. (eds.) Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, pp. 136–148. ACM (2008)
43. Solar-Lezama, A., Tancau, L., Bodík, R., Seshia, S.A., Saraswat, V.A.: Combinatorial sketching for finite programs. In: Shen, J.P., Martonosi, M. (eds.) Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, pp. 404–415. ACM (2006)
44. Soos, M., Nohl, K., Castelluccia, C.: Extending SAT solvers to cryptographic problems. In: Kullmann, O. (ed.) SAT 2009. LNCS, vol. 5584, pp. 244–257. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02777-2_24
45. Stockmeyer, L.J., Meyer, A.R.: Word problems requiring exponential time: Preliminary report. In: Aho, A.V., et al. (eds.) Proceedings of the 5th Annual ACM Symposium on Theory of Computing, Austin, Texas, USA, 30 April–2 May 1973, pp. 1–9. ACM (1973)
46. Tentrup, L.: CAQE and quabs: Abstraction based QBF solvers. J. Satisf. Boolean Model. Comput. **11**(1), 155–210 (2019)
47. Tentrup, L., Rabe, M.N.: Clausal abstraction for DQBF. In: Janota, M., Lynce, I. (eds.) SAT 2019. LNCS, vol. 11628, pp. 388–405. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-24258-9_27

48. Vizel, Y., Gurfinkel, A., Malik, S.: Fast interpolating BMC. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 641–657. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_43
49. Vizel, Y., Weissenbacher, G., Malik, S.: Boolean satisfiability solvers and their applications in model checking. Proc. IEEE **103**(11), 2021–2035 (2015)
50. Wimmer, R., Karrenbauer, A., Becker, R., Scholl, C., Becker, B.: From DQBF to QBF by dependency elimination. In: Gaspers, S., Walsh, T. (eds.) SAT 2017. LNCS, vol. 10491, pp. 326–343. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66263-3_21
51. Wimmer, R., Scholl, C., Becker, B.: The (D)QBF preprocessor hqspre - underlying theory and its implementation. J. Satisf. Boolean Model. Comput. **11**(1), 3–52 (2019)
52. Wimmer, R., Scholl, C., Wimmer, K., Becker, B.: Dependency schemes for DQBF. In: Creignou, N., Le Berre, D. (eds.) SAT 2016. LNCS, vol. 9710, pp. 473–489. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40970-2_29