What is the State-of-the-Art in DQBF solving?[*]

Gergely Kovásznai
kovasznai.gergely@uni-eszterhazy.hu

IoT Research Institute
Eszterházy Károly University
Eger, Hungary

Abstract

Dependency Quantified Boolean Formulas (DQBF) are obtained by adding Henkin quantifiers to Boolean formulas and have seen growing interest in the last years. In contrast to QBF, problem descriptions in DQBF can possibly be exponentially more succinct. An the other hand, while QBF is PSpace-complete, DQBF was shown to be NExpTime-complete.

Many practical problems are known to be NExpTime-complete and, hence, can be reduced to DQBF, e.g., partial information non-cooperative games, certain bit-vector logics and partial equivalence checking problems.

In this paper, I give a detailed survey on solving approaches for DQBF in a chronological order, as well as preprocessing and inprocessing techniques, and further advances in DQBF solving.

1 Introduction

With steadily increasing success of decision procedures for propositional formulas (SAT) and Quantified Boolean Formulas (QBF), also interest in Dependency Quantified Boolean Formulas (DQBF) has grown during the last years. DQBF has first been described in [PR79] and comprises the set of propositional formulas which are obtained by adding Henkin quantifiers [Hen61], also called branching quantifiers, to Boolean logic. In contrast to QBF, the dependencies of a variable in DQBF are explicitly specified instead of being implicitly defined by the order of the quantifier prefix. This enables us to also use partial variable orders as part of a formula instead of only allowing total ones.

As a result, problem descriptions in DQBF can possibly be exponentially more succinct. While QBF is PSpace-complete [Pap94], DQBF was shown to be NExpTime-complete [PRA01, PR79]. Aside from DQBF, many practical problems are known to be NExpTime-complete. This includes, e.g., partial information non-cooperative games [PRA01] or certain bit-vector logics [KFB12, WHdM10, KFB15] used in the context of Satisfiability Modulo Theories (SMT). Although quantifier-free bit-vector logics (QF_BV) are the essential ingredient for hardware verification, and practically efficient direct solving approaches for QF_BV exist as well as indirect ones that reduce QF_BV to other targets logics such as EPR [KFB13] or to model checking problems [FKB13],

[*]The research was supported by the grant EFOP-3.6.1-16-2016-00001 "Complex improvement of research capacities and services at Eszterhazy Karoly University".

In: E. Vatai (ed.): Proceedings of the 11th Joint Conference on Mathematics and Computer Science, Eger, Hungary, 20th – 22nd of May, 2016, published at http://ceur-ws.org

DQBF has not succeeded to be a target logic for QF_BV so far. More recently, [GRS+13a, GRS+13b] showed that *partial equivalence checking (PEC) problems* can be naturally encoded as DQBFs. Currently, solving PEC problems seems to be the “killer application” for DQBF, as we will discuss in Section 4.

For DQBF, only a few direct solving approaches exist. In Section 3, we will introduce, chronologically, the first one called *DQDPLL* [FKB12], which was followed by no publicly available implementation. In Section 5.1, another solving approach called *bounded unsatisfiability* [FT14] will be introduced, which can solve unsatisfiable DQBFs very efficiently, but cannot deal with satisfiable instances in practice. In Sections 5.2 and 5.3, we will introduce two concrete, general-purpose DQBF solvers *iDQ* [FKBV14] and *HQS* [GWR+15], respectively. While iDQ is publicly available, HQS was reported to be more efficient in practice, but, to the best of our knowledge, is not yet publicly available. We will dedicate Section 6 to the topic of DQBF *preprocessing.* In Section 7, we will summarize the most recent advances in DQBF solving.

2 Preliminaries

Let V be a set of Boolean variables. A *positive literal* is a Boolean variable $x \in V$, a *negative literal* is its negation $\overline{x}$. For a given literal l, we write $var(l)$ to reference its corresponding variable. A *clause* is a disjunction of literals. A Boolean formula is in *Conjunctive Normal Form (CNF)*, if it is a conjunction of clauses.

A *Quantified Boolean Formula (QBF)* is defined as

$$Q_1 x_1 \dots Q_n x_n \, . \, \phi \tag{1}$$

where $Q_i \in \{\forall, \exists\}$ are quantifiers, $x_i \in V$ are distinct variables, and ϕ is a Boolean formula in CNF over the variables $x_1, \dots, x_n$. We call $Q_1 x_1 \dots Q_n x_n$ the *quantifier prefix*, and ϕ the *matrix.*

A *Dependency Quantified Boolean Formula (DQBF)* is the generalization of a QBF as follows. A DQBF is defined as

$$\forall u_1 \dots \forall u_m \; \exists e_1(u_{1,1}, \dots, u_{1,k_1}) \dots \exists e_n(u_{n,1}, \dots, u_{n,k_n}) \, . \, \phi \tag{2}$$

where ϕ is a Boolean formula in CNF over the variables $u_1, \dots, u_m, e_1, \dots, e_n$. The formalism $e_i(u_{i,1}, \dots, u_{i,k_i})$ means that the existential variable e_i *depends only on* the universal variables $u_{i,1}, \dots, u_{i,k_i}$. We use $D_{e_i} := \{u_{i,1}, \dots, u_{i,k_i}\}$ to denote e_i's dependency set. Sometimes when we do not need to enumerate the exact dependencies in advance, we will write (2) as

$$\forall u_1 \dots \forall u_m \; \exists e_1(D_{e_1}) \dots \exists e_n(D_{e_n}) \, . \, \phi \tag{3}$$

Furthermore, let us extend the notion of dependency to literals, defining $D_l := D_{var(l)}$ for any literal l.

Note that in DQBF the dependencies of existential variables are always explicitly given, in contrast to QBF where an existential variable depends on all the universal variables to the left in the quantifier prefix. Thus, the QBF (1) can be considered as a special case of DQBF, where for all $Q_i = \exists$ it holds that $D_{x_i} = \{x_j \mid 1 \leq j < i,\ Q_j = \forall\}$. While in QBF the dependencies of the existential variables induce linear ordering, in DQBF this is not always the case.

A truth *assignment* is a (partial) mapping $\alpha : V \mapsto \{0, 1\}$ from the variables of a formula to truth values. Sometimes we will denote an assignment as $\alpha = \{x_1/v_1, \dots, x_k/v_k\}$ where all $x_i \in V$ and $v_i \in \{0, 1\}$. Given a formula ϕ, the term $\phi[\alpha]$ denotes the formula resulted by replacing all the variables x_i in ϕ with $\alpha(x_i)$.

A Boolean formula ϕ is *satisfiable* if and only if there exists an assignment α such that $\phi[\alpha]$ is evaluated to 1. We then call α a model of ϕ. In DQBF (as well as in QBF), a model cannot be expressed by a single assignment. Instead, we use *Skolem functions* to represent solutions of a formula. A Skolem function $f_e : \{1, 0\}^{|D_e|} \mapsto \{1, 0\}$ describes the evaluation of an existential variable e under a given assignment to its dependencies. Let ϕ_{sk} denote the formula obtained from ϕ by replacing all existential variables e by their Skolem function f_e. The DQBF (3) is satisfiable if and only if there exist Skolem functions $f_{e_1}, \dots, f_{e_n}$ such that ϕ_{sk} is satisfied by all possible assignments to the universal variables.

Effectively Propositional Logic (EPR), known as the Bernays-Schönfinkel class, is a NExpTime-complete fragment of first-order logic [Lew80]. It consists of the first-order formulas in prenex form that contain no function symbol of arity greater than 0, and no existential quantifier within the scope of a universal quantifier. After Skolemization, existential variables turn into constants (i.e., function symbols of arity 0). Consequently, an EPR atom can be defined as an expression of the form $p(t_1, \dots, t_n)$ where p is a predicate symbol of arity n, and each t_i is either a universal variable or a constant. EPR literals and clauses are defined similarly as in Boolean logic, by replacing the word “Boolean variable” with “atom”. An instance of an EPR clause C is a clause $C[\alpha]$ for some assignment α to the universal variables in C.

3 First Solving Approach – DQDPLL

The first direct solving approach for DQBF is called *DQDPLL* [FKB12], which is the adaptation of QDPLL [CGS98] being one of the most successful solving approaches for QBF. The paper shows how to adapt the building blocks of the DPLL-style approach, one by one, like unit propagation, clause learning, backtracking, universal reduction, pure literal reduction, watched literal schemes, etc.

The fundamental difference between DQDPLL and QDPLL is how decisions are saved in the decision stack: while QDPLL only has to save the decision literal l_e where e is an existential variable, DQDPLL additionally has to save a so-called *Skolem clause* linked with the current branch of the search tree represented by a partial assignment α. The Skolem clause that represents this decision l_e according to the assignments to the universal variables on which e depends:

$$C_{sk} \;:=\; \Big(l_e \vee \bigvee_{u \in D_e} l_u\Big), \text{ where } l_u = \begin{cases} u, & \text{if } \alpha(u) = 0 \\ \overline{u}, & \text{if } \alpha(u) = 1 \\ 0, & \text{if } \alpha(u) = \mathit{undef} \end{cases}$$

Unit propagation can be implemented in the same way as done for QDPLL, with the extension of adding a Skolem clause as already described.

Clause learning also works as in QDPLL, however we need to differentiate between *temporary learned clauses* and *permanent learned clauses.* Any learned clause created by resolution with at least one Skolem clause or with a temporary learned clause is only valid as long as all clauses participating in the resolution steps are still part of the formula, and, therefore, will be a temporary learned clause itself. A permanent learned clause is created when no Skolem clause and no temporary learned clause was part of the resolution process.

Although *backtracking* seems to be implemented by DQDPLL in a quite standard way, there is a non-standard solution in dealing with satisfying branches. Whenever the current clause set is satisfied by the current branch of the search tree, no backtracking takes place, thus, no decisions are undone and no Skolem clauses are removed. Consequently, whenever the current branch makes the current clause set unsatisfiable and real backtracking takes places, it might happen that the algorithm jumps back to a previous branch of the search tree. Figure 1 illustrates this particular phenomenon, where the original DQBF has an existential variable e with $D_e = \{u_2, u_3\}$ and DQDPLL has constructed the search tree shown in the figure so far. Let us suppose that the rightmost branch of the tree makes the formula unsatisfiable and, therefore, the algorithm analyses the conflict and discovers that the current assignment to $e(1,0)$ is one of the causes of the conflict and thus it is needed to be changed, e.g., from 0 to 1. Since e does not depend on u_1, it might be the case that $e(1,0)$ was assigned not on the rightmost branch, but on a previous branch which assigned the same truth values to u_2 and u_3 as the rightmost branch does. If this is the case, we need to backtrack to that particular branch.

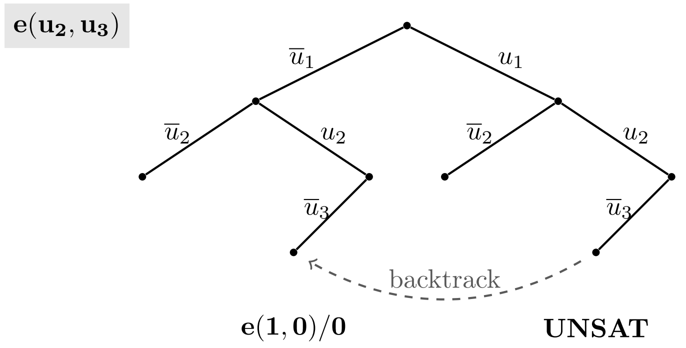

Figure 1: Example of DQDPLL backtracking to a previous branch of the search tree.

Consequently, DQDPLL sometimes needs to jump several branches back and to start to traverse those branches again. Note that for QBF this cannot happen since e would depend on all of u_1, u_2, and u_3. We think that this phenomenon is one of the most fundamental reasons why DQDPLL "does not perform very well" [FKB12] in practice, while QDPLL is considered to be quite effective.

Nevertheless, DQDPLL was a pioneer approach in DQBF solving. It addressed a lot of topics which later were taken up by others. For example, pure literal reduction for DQBF was later extended and generalized by [GWR+15, WGN+15]. Universal reduction was also used by [WGN+15]. VSIDS-based selection heuristics were applied by [FKBV14] as well. Furthermore, DQDPLL investigated the topic of clause and cube learning which has not been addressed by any other DQBF solving approaches yet.

4 First Killer Application – PEC Problems

A fundamental application for DQBF is partial, or imperfect, information non-cooperative games [PR79]. In the context of Satisfiability Modulo Theories (SMT), certain bit-vector logics provide applications for DQBF, since they have the same computational complexity [KFB12, WHdM10, KFB15] as DQBF does. Although quantifier-free bit-vector logics (QF_BV) are the essential ingredient for hardware verification, and efficient direct solving approaches for QF_BV exist as well as indirect ones that reduce QF_BV to other targets logics such as EPR [KFB13] or to model checking problems [FKB13], DQBF has not succeeded to be a target logic for QF_BV so far.

[GRS+13a, GRS+13b] show that *partial equivalence checking problems* can be naturally encoded as DQBFs. Equivalence checking is about to decide if two combinational circuits always produce the same outputs from the same inputs. If one of the circuits is not completely specified, but contains missing parts, so-called black boxes, then the problem is called partial equivalence checking (PEC) as shown in Figure 2. PEC asks the question if there exist implementations of all the black boxes such that the two circuits become equivalent.

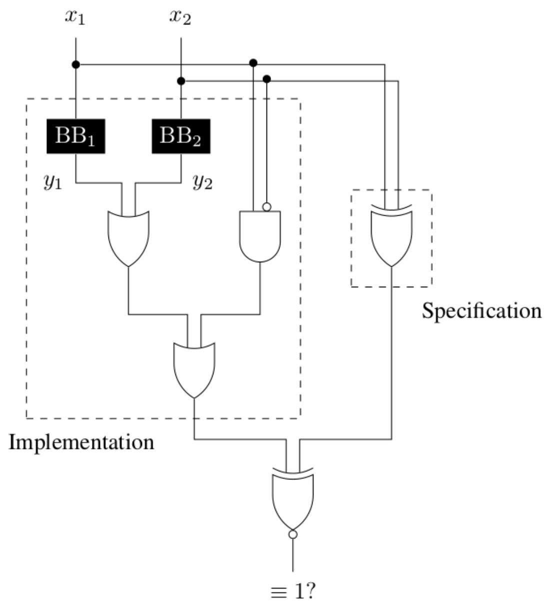

Figure 2: Example of a PEC problem [GRS+13b].

The inputs of the combined circuit are modeled as universal variables and the outputs of the black boxes as existential variables. Let us introduce the following notations:

- $\vec{X}$ are the inputs of the circuit
- $\vec{I_i}$ are the inputs of the ith black box ($1 \le i \le m$)
- $\vec{Y_i}$ are the outputs of the ith black box
- $\vec{F_i}(\vec{X}, \vec{Y_1}, \dots, \vec{Y}_{i-1})$ are the functions defining $\vec{I_i}$
- $\vec{R}(\vec{X}, \vec{Y_1}, \dots, \vec{Y_m})$ are the outputs of the circuit

Using these notations, the following DQBF encodes the PEC problem:

$$\forall \vec{X}\ \forall \vec{I_1} \dots \forall \vec{I_m}\ \exists \vec{Y_1}(\vec{I_1}) \dots \exists \vec{Y_m}(\vec{I_m})\ \exists \vec{A}(\vec{X}, \vec{I_1}, \dots, \vec{I_m})\ .\ \phi' \tag{4}$$

where ϕ' is a CNF of the following Boolean formula:

$$\phi \;:=\; \left(\vec{I}_1 \not\equiv \vec{F}_1(\vec{X})\right) \vee \cdots \vee \left(\vec{I}_m \not\equiv \vec{F}_m(\vec{X}, \vec{Y}_1, \ldots, \vec{Y}_{m-1})\right) \vee \vec{R}(\vec{X}, \vec{Y}_1, \ldots, \vec{Y}_m)$$

In (4), $\vec{A}$ denotes the auxiliary variables introduced by the Tseitin transformation in order to obtain from ϕ a CNF ϕ' of linear size.

5 Solvers

5.1 Approximation and Unrolling

DQBF generalizes QBF by allowing non-linear dependencies between variables. One could generate from the original DQBF a *QBF approximation* and then to apply a QBF solver, one of the several efficient ones like DepQBF [LB10] and RAReQS [JKMSC12]. Such a method might give non-precise results for the original DQBF though, due to QBF not being able to capture certain dependencies.

The approach of *"bounded unsatisfiability"* [FT14] proposes an efficient and precise iterative QBF approximation technique that focuses on the refutation of a DQBF. Given a DQBF Φ and a bound $k \geq 1$, we construct a QBF Ψ_k by using k copies $v^1, \ldots, v^k$ of each variable v in the quantifier prefix and, similarly, k copies of the matrix, in which every variable v is replaced by $v^1, \ldots, v^k$, respectively. The approach is a refutation technique which means to negate Ψ_k meanwhile, and that the satisfiability of Ψ_k implies that Φ is unsatisfiable.

In order to ensure the result to be precise, we also need to specify a consistency condition as a guard that enforces the originally existential variables act according to the original dependencies defined in Φ.

Let (3) be the original DQBF. Given a bound $k \geq 1$, the resulting QBF approximation looks as follows:

$$\Psi_k \;:=\; \exists u_1^1 \ldots \exists u_1^k \exists u_2^1 \ldots \exists u_m^k \; \forall e_1^1 \ldots \forall e_1^k \forall e_2^1 \ldots \forall e_n^k \; . \; \bigwedge_{1 \leq i \leq n} \mathit{consistent}(e_i, k) \Rightarrow \bigvee_{1 \leq i \leq k} \overline{\phi}\underbrace{[v/v^i]}_{\forall v \in V}$$

Note that the copies of the matrix are negated, as well as the entire quantifier prefix, which results in the type of the quantifiers being flipped. Note furthermore that in Ψ_k all the existential variables are in front of the quantifier prefix, therefore they do not depend on any of the universal variables, thus their assignments are not restricted in any way. This is why we need the consistency conditions to add which are represented as the following Ackermann constraints:

$$\mathit{consistent}(e, k) \;:=\; \bigwedge_{1 \leq i,j \leq k} \Big(\bigwedge_{u \in D_e} u^i = u^j \;\Rightarrow\; e^i = e^j \Big)$$

Note that, for the sake of readability, the formulas above are not in CNF, but it is easy to transform them to CNF.

Φ is unsatisfiable if and only if there exists a bound $k \geq 1$ such that Ψ_k is satisfiable. In practice, the best way to run the approach is to start with $k = 1$. If Ψ_k is unsatisfiable, then increase the bound to $k := k + 1$ and check the satisfiability of the resulting Ψ_k.

As the experiments in [FT14] show, the approach works fast on PEC benchmarks. This is greatly due to the fact that for more than 94% of the PEC instances it is sufficient to use bound $k = 2$. However, k is proportional to the number of black boxes in a PEC instance.

The downside of the approach is that it works in practice only for unsatisfiable DQBFs. For proving a DQBF to be satisfiable, we need to iteratively increase k up to 2^m, where m is the number of universal variables, and to check the satisfiability of the resulting QBF in every iteration, which makes this approach infeasible for most of the satisfiable DQBF benchmark instances.

5.2 Instantiation – iDQ

Effectively Propositional Logic (EPR) is NExpTime-complete [Lew80], i.e., it has the same computational complexity as DQBF. Consequently, it is possible to use EPR solvers, e.g., iProver [Kor08] being the currently most successful one, to solve DQBF given some polynomial translation from DQBF to EPR. The translation from QBF to EPR described in [SLB12] can be extended to DQBF easily [FKBV14], as shown for the following DQBF:

$$\Phi \;:=\; \forall u_1 \forall u_2 \; \exists e_1(u_1, u_2) \exists e_2(u_2) \; . \; (u_1 \vee e_1) \wedge (\overline{u}_2 \vee \overline{e}_1 \vee e_2) \tag{5}$$

The translation replaces each existential variable with its Skolem function, furthermore, as a technical step, replaces each universal variable u with $p(u)$ where p is a fixed predicate, and adds the constraints $p(1)$ and $\overline{p}(0)$. Thus, the resulting EPR formula looks as follows:

$$\forall u_1 \forall u_2\ .\ \big(p(u_1) \vee e_1(u_1, u_2)\big)\ \wedge\ \big(\overline{p}(u_2) \vee \overline{e}_1(u_1, u_2) \vee e_2(u_2)\big)\ \wedge\ p(1) \wedge \overline{p}(0)$$

However, since EPR solvers in general have to reason with predicates and larger domains, solvers directly working on the Boolean level should have an advantage in solving DQBF. [FKBV14] proposes an instantiation-based approach to solving DQBF directly, which is closely related to the so-called *Inst-Gen calculus* [Kor13], the underlying decision procedure in iProver [Kor08]. The resulting solver *iDQ* [FKBV14] is the first publicly available DQBF solver.

Similar to the Inst-Gen calculus and some instantiation-based QBF solving approaches such as [Ben04, JKMSC12, JMS13], iDQ generates instances of the input clauses in a sophisticated way. A clause instance is generated by assigning truth values to some of the universal variables in the clause. Given a clause C and literals $l_1, \dots, l_k$, an instance of C is denoted by $C_{l_1 \dots l_k}$, as an alternative notation for $C'[var(l_1)/v_1, \dots, var(l_k)/v_k]$ where $v_i = 0$ resp. $v_i = 1$ if l_i is negative resp. positive, and C' is the EPR counterpart of C.

Given a DQBF Φ, iDQ starts the solving with an initial set of clause instances, which contains one instance for each input clause. An initial instance is generated by the so-called *unique minimal instantiation* that removes from the clauses all the universal literals by assigning them to 0. From (5) we get the following initial instances:

$$(e_1)_{\overline{u}_1}\ \wedge\ (\overline{e}_1 \vee e_2)_{u_2} \tag{6}$$

We then create a *Boolean over-approximation* of the clause instance set by mapping the variable instances to new Boolean variables. By doing so, we need to respect the dependencies between variables in Φ and to map the different variable instances to different Boolean variables. The Boolean over-approximation of (6) is as follows:

$$(x_1) \wedge (\overline{x}_2 \vee x_3) \tag{7}$$

where x_1, x_2, x_3 are the new Boolean variables. Note that the same existential variable e_1 is mapped to two different Boolean variables: the instance $(e_1)_{\overline{u}_1}$ is mapped to x_1, and $(e_1)_{u_2}$ to x_2. Now we need to check if (7) is satisfiable by using any off-the-shelf SAT solver. If the SAT solver told us that (7) was unsatisfiable, iDQ would terminate and claim Φ to be unsatisfiable. However, (7) is satisfiable and the SAT solver could return a satisfying assignment $\alpha = \{x_1/1, x_2/0, x_3/0\}$.

We now check, by applying *unification*, whether α is a valid satisfying assignment for (6). This is the case if and only if no pair of oppositely signed, (selected) satisfying literals corresponds to the same existential variable such that the EPR counterparts of the two literal instances can be unified. In our example, α is indeed not a valid assignment: x_1 and $\overline{x}_2$ are selected as satisfying literals, both correspond to e_1, and their EPR counterparts $e_1(0, u_2)$ and $e_1(u_1, 1)$ can be unified. We use the most general unifier for adding two new clause instances as follows:

$$(e_1)_{\overline{u}_1} \wedge (e_1)_{\overline{u}_1 u_2} \wedge (\overline{e}_1 \vee e_2)_{u_2} \wedge (\overline{e}_1 \vee e_2)_{\overline{u}_1 u_2}$$

The Boolean over-approximation is now given by:

$$(x_1) \wedge (x_2) \wedge (\overline{x}_3 \vee x_4) \wedge (\overline{x}_2 \vee x_4) \tag{8}$$

Note that e_2 is mapped to the same variable x_4 in both clause instances since e_2 does not depend on u_1 in Φ. (8) is satisfiable and the SAT solver could return the satisfying assignment $\alpha = \{x_1/1, x_2/1, x_3/0, x_4/1\}$. Now, α is valid since we can select a satisfying literal in each clause such that no unification resulting in new instances can be applied to them. Therefore, the algorithm terminates and Φ is proven to be satisfiable.

[FKBV14] proposes a lot of implementation optimizations. Instantiation, unification, and redundancy checks are all implemented by only using bit-vectors for the sake of taking advantage of the Boolean domain. Also because of the Boolean domain, iDQ is able to apply VSIDS heuristics that come from SAT solving. The reported experiments show that iDQ outperforms iProver on the DQBF benchmarks.

5.3 Elimination – HQS

Similar to the approach in Section 5.1, from DQBF a QBF approximation is generated by the solving technique in [GWR+15]. The resulting DQBF solver called *HQS* uses an elimination-based strategy accompanied with several optimizations. The strategy eliminates a minimum set of variables that cause the non-linear dependencies in the DQBF. Once there are only linear dependencies left, an off-the-shelf QBF solver can be applied for the remaining (QBF) formula. A lot of components of the approach rely on And-Inverter-Graphs (AIGs) [PS09].

HQS's approach is illustrated in Figure 3. First, some preprocessing steps known from QBF preprocessing and adapted to DQBF are applied. (We will dedicate Section 6 to the topic of DQBF preprocessing.) Gate detection counts as a further preprocessing step, since the detected gates are used later by AIG operations.

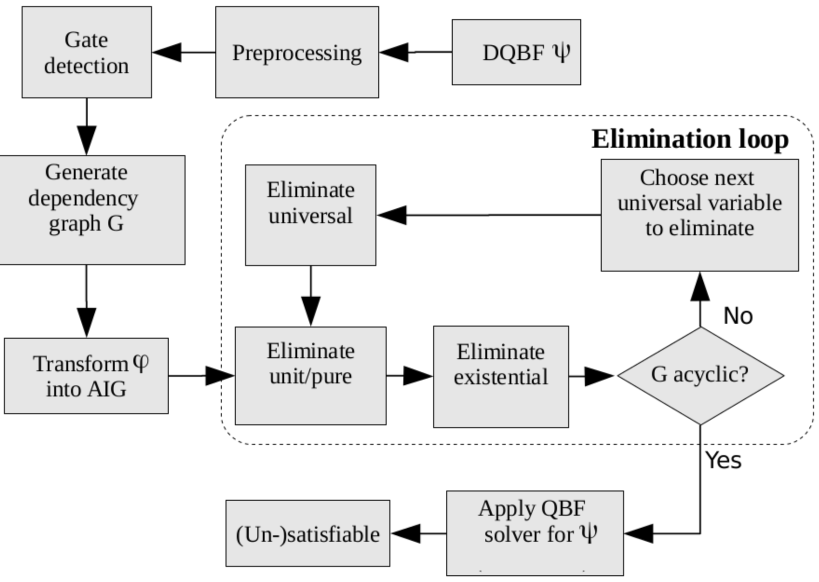

Figure 3: Algorithmic flow of HQS [GWR$^+$15].

[GWR$^+$15] proposes a novel technique for observing if there are only linear dependencies left in the DQBF. It is based on the so-called *dependency graph*, which is generated from the original DQBF. The vertices of the dependency graph are the existential variables, and an edge connects e_1 with e_2 if and only if e_1 depends on some variable on which e_2 does not depend. The dependency graph is constantly updated during the solving process until it becomes acyclic, which shows that there are only linear dependencies left.

As an inprocessing step, *unit and pure variables* are eliminated in each iteration. HQS employs a rather cheap syntactic check for unit and pure literals based on the AIG.

The main operation in each iteration is the elimination of existential and universal variables. The *elimination of an existential variable* is the less expensive operation because HQS eliminates only the innermost existential variables in the quantifier prefix, i.e., those ones that depend on all the universal variables. Given the DQBF (3), the existential variable e_i can be eliminated if $D_{e_i} = \{u_1, \dots, u_m\}$. In this case, the matrix ϕ is duplicated such that the two duplicates replace e_i with 0 and 1, respectively:

$$\forall u_1 \dots \forall u_m \; \exists e_1(D_{e_1}) \dots \exists e_{i-1}(D_{e_{i-1}}) \exists e_{i+1}(D_{e_{i+1}}) \dots \exists e_n(D_{e_n}) \; . \; \phi[e_i/0] \;\vee\; \phi[e_i/1] \tag{9}$$

When *eliminating a universal variable* u_i, all the existential variables e_j that depend on u_i must be duplicated. Furthermore, the matrix ϕ must be duplicated as $\phi[u_i/0]$ and $\phi[u_i/1]$, similar to what we did in (9). However, $\phi[u_i/0]$ must use the original existential variables e_j, while $\phi[u_i/1]$ must use the duplicates e'_j.

$$\forall u_1 \dots \forall u_{i-1} \forall u_{i+1} \dots \forall u_m \; \exists e_1\big(D_{e_1} \setminus \{u_i\}\big) \dots \exists e_n\big(D_{e_n} \setminus \{u_i\}\big) \; \underbrace{\exists e'_j\big(D_{e_j} \setminus \{u_i\}\big)}_{\forall e_j \;.\; u_i \in D_{e_j}} \; . \tag{10}$$
$$\phi[u_i/0] \;\wedge\; \phi[u_i/1, \;\; \underbrace{e_j/e'_j}_{\forall e_j \;.\; u_i \in D_{e_j}} \;\;]$$

The elimination of a universal variable is rather expensive, this is why the next universal variable to eliminate is chosen by some heuristics. First, the approach determines the minimal set of universal variables whose elimination leads to a QBF problem. The problem of finding this minimal set is expressed as a MaxSAT problem as explained in [GWR$^+$15]. Then, these universal variables are ordered according to the number of the existential variables which would be duplicated in (10).

The experiments reported in [GWR$^+$15] compare HQS against iDQ. The results show that HQS is able to solve 50% more benchmark instances than iDQ and speeds up the computation time by up to four orders of magnitude. Unfortunately, HQS is not publicly available.

6 Preprocessing

In practice, it is crucial to apply preprocessing to the input formula before it is passed to an actual solver. Preprocessing techniques often reduce the computation time of solvers by orders of magnitude. For QBF, there exist efficient preprocessing tools in practice like sQueezeBF [GMN10] and bloqqer [BLS11]. The paper [WGN$^+$15] proposes several preprocessing techniques for DQBF, as a continuation of the work has been started in [GWR$^+$15], and reports successful experiments with the solvers iDQ and HQS. All of these preprocessing techniques are the generalizations of known preprocessing techniques for SAT and QBF.

Universal reduction is a well-known simplification technique for QBF that can be easily adapted to DQBF. In any clause C, if there is a universal literal $l \in C$ such that no other literal $k \in C$ depends on l, then l can be removed from C. The condition of k not depending on l can be formalized as $var(l) \notin D_k$.

Another technique that comes from QBF solving is *eliminating an existential variable by resolution.* By applying all the possible resolution steps to the matrix on an existential variable e, one can eliminate e from the formula completely. However, there is a necessary condition for applying this preprocessing step: in all clauses C that contain e, all the literals $k \in C$ must only depend on variables on which e depends, i.e., $D_k \subseteq D_e$. Note however that it might be rather expensive to apply all the possible resolution steps since the number of clauses might grow considerably. Therefore it is necessary to apply some cost function, e.g., the one defined in [WGN$^+$15], in order to decide if it is worth to eliminate a particular existential variable in this way.

Unit and pure literals can be replaced by the constants 0 or 1 without influencing the formula's truth value. For DQBF, it is worth to generalize unit and pure literals by introducing the concepts of *backbones* and *monotonic variables*, respectively. In a Boolean formula ϕ, a variable v is a backbone if (B0) $\phi[v/0]$ or (B1) $\phi[v/1]$ is unsatisfiable. A variable v is a monotonic if (M0) $\phi[v/0] \wedge \overline{\phi}[v/1]$ or (M1) $\phi[v/1] \wedge \overline{\phi}[v/0]$ is unsatisfiable. Note that the checks of a variable being backbone or monotonic can be done by a SAT solver on the matrix of the DQBF, therefore they count as cheap checks. Of course, we also need to define what to do with the DQBF Φ if it turned out v was a backbone or monotonic:

- If v is a universal variable:
  - In case of (B0) or (B1), Φ is unsatisfiable.
  - In case of (M0), eliminate v by replacing with 0.
  - In case of (M1), eliminate v by replacing with 1.
- If v is an existential variable:
  - In case of (B0) or (M0): eliminate v by replacing with 1.
  - In case of (B1) or (M1): eliminate v by replacing with 0.

Blocked clauses are important to detect in both SAT and QBF solving [JBH10, BLS11], since blocked clauses can be removed from a formula without changing its truth value. For DQBF, a literal l blocks a clause C with $l \in C$ if for all other clauses C' with $\overline{l} \in C'$

- the resolvent of C and C' on l is tautological, i.e., it contains both k and $\overline{k}$ for some variable k, and
- $D_k \subseteq D_l$.

Since removing blocked clauses makes the original formula shorter, one might want to increase the chance of finding blocked clauses by temporarily extending the clauses with so-called hidden and covered literals as detailed in [GWR$^+$15].

In QBF solving it usually pays off to *reduce the dependency sets* of variables [LB09, Gel11, SS12]. The concept can be easily adapted to DQBF. Even if $u \in D_e$ for some universal variable u and some existential variable e, due to the structure of the matrix, it might turn out that the Skolem function of e does not use u's value at all to satisfy the formula, therefore u can be removed from D_e. Since this semantic check has the same complexity as DQBF solving itself, sufficient syntactic criteria called *dependency schemes* are applied instead. In [GWR+15], the co-called *standard dependency scheme* is defined for DQBF. From the input formula an undirected graph is constructed which contains all the formula's variables v and clauses C as vertices, and creates an edge from v to C if v appears in C. The standard dependency scheme is defined as follows: an existential variable e does not depend on a universal variable u if there is no path in the graph between u and e, visiting only such existential variables e' that $u \in D_{e'}$.

An interesting choice for a preprocessing step is the QBF approximation technique using unrolling by [FT14], which we have already described in Section 5.1. In the experiments in [GWR+15], this unrolling technique with a bound $k = 1$ and $k = 2$ is used.

The experiments in [GWR+15] apply various preprocessing techniques, including the ones above, in several combinations. Interestingly, the results show that the preprocessor alone is able to solve a considerable amount of benchmark instances. On the remaining instances, the DQBF solvers iDQ and HQS were executed, and the results show that preprocessing increases the number of solved instances by a factor of ca. 2.5. Computation time often drops by orders of magnitude.

7 Further Advances in DQBF Solving

A very recent paper [WSWB16] investigates if the *dependency schemes* for QBF could be lifted to DQBF. Note that the standard dependency scheme has already been addressed by [WGN+15], cf. Section 6. Other kinds of dependency schemes are also known for QBF, therefore [WSWB16] investigates how to adapt them to DQBF and checks if the resulting dependency schemes are sound. The overall picture of those dependency schemes can be seen in Figure 4, where the ones inside the blue box are proved to be sound for DQBF, while the ones outside of the blue box to be unsound. Without getting into details, all the different variants of the standard dependency scheme, whose notations start with 's', are sound, which shows that [WGN+15] has chosen a proper dependency scheme for preprocessing, cf. Section 6. It is also interesting that the reflexive dependency schemes, whose notations start with 'r', are sound as well. From the soundness point of view, it does not matter if one picks either a quadrangle or a triangle dependency scheme, and, similarly, if one picks resolution path connectedness between clauses instead of the usual notion of connectedness.

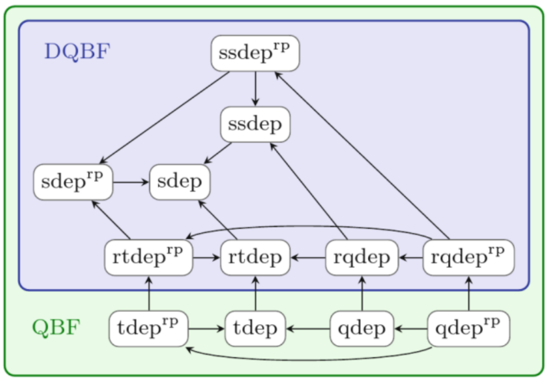

Figure 4: Different dependency schemes for DQBF [WSWB16]. Those outside of the blue box are not sound.

Even if DQBF solvers like iDQ and HQS can tell us whether a DQBF is satisfiable, in practice we also need to *synthesize Skolem functions* for the existential variables. From this point of view, DQBF certification is investigated in [BCJ13]. Since Skolem functions could be obtained from resolution proofs, the paper presents a kind of negative result though: the DQBF-version of Q-resolution is incomplete.

A very recent paper [BCSS16] investigates how the different resolution systems for QBF can be lifted to DQBF, and if the lifted variants are sound and/or complete. The overall picture of those resolution calculi can be seen in Figure 5. Without getting into details, Q-resolution (Q-Res), which is the best-studied resolution system for QBF, is incomplete for DQBF, as it was already shown by [BCJ13]. From the completeness point of view, it does not matter if resolution steps are also allowed on universal variables (QU-Res). Interestingly, if long-distance resolution is applied (LD-Q-Res, LQU$^+$-Res), the resulting calculi become unsound. However, there exist instantiation-based calculi that are sound and complete for DQBF: ∀Exp+Res can be considered a very simple instantiation-based calculus, which IR-calc extends with partial assignments. IRM-calc combines IR-calc with solutions from LD-Q-Res which make the calculus unsound.

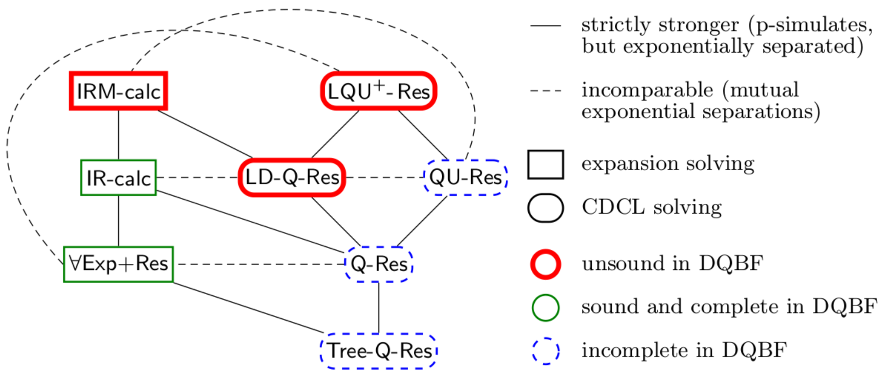
IRM-calc
LQU+-Res
IR-calc
LD-Q-Res
QU-Res
∀Exp+Res
Q-Res
Tree-Q-Res
strictly stronger (p-simulates, but exponentially separated)
incomparable (mutual exponential separations)
expansion solving
CDCL solving
unsound in DQBF
sound and complete in DQBF
incomplete in DQBF

Figure 5: Resolution systems lifted to DQBF and their soundness/completeness [BCSS16].

8 Conclusion

In this paper, we gave a thorough survey on the state-of-the-art of DQBF solving. We reported the advent of the first practically efficient solving approaches and solver implementations, such as iDQ and HQS. Since, in practice, it is crucial to apply preprocessing before solving the actual formula, it is good news that a powerful DQBF preprocessor was implemented recently. DQBF solving is an emerging area as shown by the recent advances that we summarize at the end of the paper.

References

[BCJ13] V. Balabanov, H. K. Chiang, and J. R. Jiang. Henkin quantifiers and boolean formulae: A certification perspective of DQBF. *Theoretical Computer Science*, 2013.

[BCSS16] O. Beyersdorff, L. Chew, R. A. Schmidt, and M. Suda. Lifting qbf resolution calculi to DQBF. In *Proc. SAT 2016*, volume 9710 of *LNCS*, pages 490–499, 2016.

[Ben04] M. Benedetti. Evaluating QBFs via symbolic skolemization. In *Proc. LPAR 2004*, pages 285–300, 2004.

[BLS11] A. Biere, F. Lonsing, and M. Seidl. Blocked clause elimination for QBF. In *Proc. CADE 2011*, volume 6803 of *LNCS*, pages 101–115, 2011.

[CGS98] M. Cadoli, A. Giovanardi, and M. Schaerf. *Algorithm to evaluate Quantified Boolean Formulae*, pages 262–267. 1998.

[FKB12] A. Fröhlich, G. Kovásznai, and A. Biere. A DPLL algorithm for solving DQBF. In *Proc. POS 2012*, 2012.

[FKB13] A. Fröhlich, G. Kovásznai, and A. Biere. Efficiently solving bit-vector problems using model checkers. In *Proc. SMT 2013*, pages 6–15, 2013.

[FKBV14] A. Fröhlich, G. Kovásznai, A. Biere, and H. Veith. iDQ: Instantiation-based DQBF solving. In *Proc. POS 2014, aff. to SAT 2014*, pages 103–116, 2014.

[FT14] B. Finkbeiner and L. Tentrup. Fast DQBF refutation. In *Proc. SAT 2014*, volume 8561 of *LNCS*, pages 243–251, 2014.

[Gel11] A. Van Gelder. Variable independence and resolution paths for quantified boolean formulas. In *Proc. CP 2011*, volume 6876 of *LNCS*, pages 789–803, 2011.

[GMN10] E. Giunchiglia, P. Marin, and M. Narizzano. squeezebf: An effective preprocessor for QBFs based on equivalence reasoning. In *Proc. SAT 2010*, volume 6803 of *LNCS*, pages 85–98, 2010.

[GRS$^+$13a] K. Gitina, S. Reimer, M. Sauer, R. Wimmer, C. Scholl, and B. Becker. Equivalence checking for partial implementations revisited. In *Proc. MBMV 2013*, pages 61–70, 2013.

[GRS$^+$13b] K. Gitina, S. Reimer, M. Sauer, R. Wimmer, C. Scholl, and B. Becker. Equivalence checking of partial designs using dependency quantified boolean formulae. In *Proc. ICCD 2013*, pages 396–403, 2013.

[GWR$^+$15] K. Gitina, R. Wimmer, S. Reimer, M. Sauer, C. Scholl, and B. Becker. Solving DQBF through quantifier elimination. In *Proc. DATE 2015*, pages 1617–1622. EDA Consortium, 2015.

[Hen61] L. Henkin. Some remarks on infinitely long formulas. In *Infinistic Methods*, pages 167–183. Pergamon Press, 1961.

[JBH10] M. Järvisalo, A. Biere, and M. Heule. *Blocked Clause Elimination*, volume 6015 of *LNCS*, pages 129–144. 2010.

[JKMSC12] M. Janota, W. Klieber, J. Marques-Silva, and E. Clarke. Solving QBF with counterexample guided refinement. In *Proc. SAT 2012*, volume 7317 of *LNCS*, pages 114–128, 2012.

[JMS13] M. Janota and J. Marques-Silva. On propositional QBF expansions and Q-resolution. In *Proc. SAT 2013*, volume 7962 of *LNCS*, pages 67–82, 2013.

[KFB12] G. Kovásznai, A. Fröhlich, and A. Biere. On the complexity of fixed-size bit-vector logics with binary encoded bit-width. In *Proc. SMT 2012*, 2012.

[KFB13] G. Kovásznai, A. Fröhlich, and A. Biere. BV2EPR: a tool for polynomially translating quantifier-free bit-vector formulas into EPR. In *Proc. CADE 2013*, volume 7898 of *LNAI*, pages 443–449, 2013.

[KFB15] G. Kovásznai, A. Fröhlich, and A. Biere. Complexity of fixed-size bit-vector logics. *Theory of Computing Systems*, pages 1–54, 2015.

[Kor08] K. Korovin. iProver – an instantiation-based theorem prover for first-order logic (system description). In *Proc. IJCAR 2008*, 2008.

[Kor13] K. Korovin. Inst-Gen - a modular approach to instantiation-based automated reasoning. In *Programming Logics*, pages 239–270, 2013.

[LB09] F. Lonsing and A. Biere. A compact representation for syntactic dependencies in QBFs. In *Proc. SAT 2009*, volume 5584 of *LNCS*, pages 398–411, 2009.

[LB10] F. Lonsing and A. Biere. Depqbf: A dependency-aware QBF solver. *JSAT*, 7(2-3):71–76, 2010.

[Lew80] H. R. Lewis. Complexity results for classes of quantificational formulas. *Journal of Computer and System Sciences*, 21(3):317–353, 1980.

[Pap94] C. H. Papadimitriou. *Computational complexity.* Addison-Wesley, 1994.

[PR79] G. L. Peterson and J. H. Reif. Multiple-person alternation. In *Proc. FOCS 2079*, pages 348–363, 1979.

[PRA01] G. Peterson, J. Reif, and S. Azhar. Lower bounds for multiplayer noncooperative games of incomplete information, 2001.

[PS09] F. Pigorsch and C. Scholl. Exploiting structure in an AIG based QBF solver. In *Proc. DATE 2009*, pages 1596–1601, 2009.

[SLB12] M. Seidl, F. Lonsing, and A. Biere. QBF2EPR: A tool for generating EPR formulas from QBF. In *Proc. PAAR 2012*, 2012.

[SS12] F. Slivovsky and S. Szeider. Computing resolution-path dependencies in linear time ,. In *Proc. SAT 2012*, volume 7317 of *LNCS*, pages 58–71, 2012.

[WGN$^+$15] R. Wimmer, K. Gitina, J. Nist, C. Scholl, and B. Becker. Preprocessing for DQBF. In *Proc. SAT 2015*, volume 9340 of *LNCS*, pages 173–190, 2015.

[WHdM10] C. M. Wintersteiger, Y. Hamadi, and L. de Moura. Efficiently solving quantified bit-vector formulas. In *Proc. FMCAD 2010*, 2010.

[WSWB16] R. Wimmer, C. Scholl, K. Wimmer, and B. Becker. Dependency schemes for DQBF. In *Proc. SAT 2016*, volume 9710 of *LNCS*, pages 473–489, 2016.