# iDQ: Instantiation-Based DQBF Solving*

Andreas Fröhlich[1], Gergely Kovásznai[2], Armin Biere[1], and Helmut Veith[2]

[1] Johannes Kepler University, Linz, Austria
[2] Vienna University of Technology, Wien, Austria

**Abstract**

Dependency Quantified Boolean Formulas (DQBF) are obtained by adding Henkin quantifiers to Boolean formulas and have seen growing interest in the last years. Since deciding DQBF is NExpTime-complete, efficient ways of solving it would have many practical applications. Still, there is only few work on solving this kind of formulas in practice. In this paper, we present an instantiation-based technique to solve DQBF efficiently. Apart from providing a theoretical foundation, we also propose a concrete implementation of our algorithm. Finally, we give a detailed experimental analysis evaluating our prototype iDQ on several DQBF as well as QBF benchmarks.

## 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 [33] and comprises the set of propositional formulas which are obtained by adding Henkin quantifiers [17] 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 [31], DQBF was shown to be NExpTime-complete [32, 33]. Aside from DQBF, many practical problems are known to be NExpTime-complete. This includes, e.g., partial information non-cooperative games [32] or certain bit-vector logics [25, 38] used in the context of Satisfiability Modulo Theories (SMT). More recently, also applications in the area of equivalence for partial implementations [15, 16] and synthesis for fragments of linear temporal logic [9] have been discussed and translations to DQBF have been proposed.

There has been theoretical work on succinct formalizations using DQBF and certain subclasses, e.g., DQBF-Horn has been shown to be solvable in polynomial time [7]. However, apart from our previous work on adapting DPLL for DQBF [14] and a recent incomplete approach (only allowing refutation of unsatisfiable formulas) [13], there have not been many attempts to solve DQBF problems in practice nor actual implementations of decision procedures for DQBF. As already pointed out in [14], our previous approach did not end up being very efficient. Apart from this, formula expansion and transformations specific to QBF have been discussed in [1, 2], which stayed only on the theoretical side but can yield an expansion-based DQBF solver similar to those existing for QBF [4]. In [13], an expansion-based solver is also briefly mentioned. A (not publicly available) expansion-based solver was used in [16]. Further, in [1, 2], it has been conjectured that QBF solvers based on Skolemization [3] could easily be adapted for DQBF.

*Supported by NFN S11403-N23/S11408-N23 (RiSE), Austrian Science Fund (FWF), and ICT10-050 (PROS-

However, the current implementation of the described QBF solver sKizzo [3] does not solely use Skolemization but also relies on an additional top-level DPLL approach for larger formulas. Adapting this kind of approach is not straightforward but requires special techniques as described in our previous work [14] and might have a similar negative impact on the performance of the resulting solver.

Effectively Propositional Logic (EPR) is another logic which is NExpTime-complete [26]. This implies that there exist polynomial reductions from DQBF to EPR and vice versa. Thus, it is possible to use EPR solvers, e.g., iProver [22] being the currently most successful one, to solve DQBF given some translation from DQBF to EPR. In [35], a translation from QBF to EPR is described which can be extended to DQBF easily. However, since EPR solvers in general have to reason with predicates and larger domains, solvers directly working on the propositional level should have an advantage if a DQBF formalization of a problem is more natural.

In the following, we present an instantiation-based approach to solving DQBF. Our approach is closely related to the so-called Inst-Gen calculus [23, 24], which can be considered as the state-of-the-art decision procedure for EPR [22]. While DQBF can be translated to EPR, we focus on applying the decision procedure directly on the given input logic. This results in a simpler framework and an algorithm which is easy to implement and adapt. At the same time, our approach can also be applied to QBF without further modifications. After defining some preliminaries in Sect. 2 and giving related work in Sect. 3, we provide the theoretical foundation in Sect. 4 and point out parallel features used in EPR solving. We also propose a concrete implementation of our algorithm in Sect. 5, and provide detailed experiments, comparing our prototype iDQ with state-of-the-art solvers on several DQBF as well as QBF benchmarks in Sect. 6. It turns out that our implementation results in an efficient DQBF solver that works on practical benchmarks and is even able to compete with QBF solvers on some problems. We conclude and propose directions for future work in Sect. 7.

## 2 Preliminaries

Let $V$ be a set of propositional variables. A *literal* $l$ is a variable $x \in V$ or its negation $\overline{x}$. For a given literal $l$, we write $var(l)$ to reference the corresponding variable. A *clause* $C$ is a disjunction of literals. A propositional formula $\phi$ is in *conjunctive normal form (CNF)*, if it is a conjunction of clauses. Any DQBF can always be expressed as

$$\psi \equiv Q.\phi \equiv \forall u_1, \ldots, u_m \exists e_1(u_{1,1}, \ldots, u_{1,k_1}), \ldots, e_n(u_{n,1}, \ldots, u_{n,k_n}).\phi$$

with $Q$ being the quantifier prefix and $\phi$ being a propositional formula (matrix) in CNF over the variables $V := U \cup E$ and $U = \{u_1, \ldots, u_m\}$, $E = \{e_1, \ldots, e_n\}$, $u_{i,j} \in U$, $\forall\ i \in \{1, \ldots, n\}, j \in \{1, \ldots, k_i\}$. We refer to the elements of $U$ and $E$ as the *universal variables* and *existential variables* of $\psi$, respectively. In DQBF, existential variables can always be placed after all universal variables in the quantifier prefix, since the dependencies of a certain variable are explicitly given and not implicitly defined by the order of the prefix (in contrast to QBF).

Given an existential variable $e_i$, we use $dep(e_i) := \{u_{i,1}, \ldots, u_{i,k_i}\}$ to denote its dependencies. For universal variables $u$, we define $dep(u) := \emptyset$. We extend the notion of dependency to literals, defining $dep(l) := dep(var(l))$ for any literal $l$. Obviously, any QBF $\psi_{qbf}$ can be translated to some $\psi_{dqbf}$ in the specified form by moving all universal variables to the beginning and

EED), Vienna Science and Technology Fund (WWTF).

then setting $dep(e) = \{u \in U \mid u \textit{ is before } e \textit{ in the quantifier prefix of } \psi_{qbf}\}$ for all existential variables.

An *assignment* is a (partial) mapping $\alpha : V \to \{1, 0\}$ from the variables of a formula to truth values. To simplify the notation, we extend the definition of assignments to literals, clauses and formulas in the natural way. In the rest of this paper, $\alpha(l)$, $\alpha(C)$, or $\alpha(F)$ will denote the truth value (under the assignment $\alpha$) of a literal $l$, a clause $C$, or a formula $F$, respectively. An assignment $\alpha$ to a formula $F$ is satisfying, if and only if $\alpha(F) = 1$.

A propositional formula $\phi$ in CNF is satisfiable, if and only if all clauses in $\phi$ are satisfied by at least one assignment $\alpha$. We then call $\alpha$ a *model* of $\phi$. In DQBF (as well as in QBF), a model can not be expressed by a single assignment. Instead, we use Skolem functions to represent solutions of a formula. A Skolem function $f_e : \{1, 0\}^{|dep(e)|} \to \{1, 0\}$ describes the evaluation of an existential variable $e$ under a given assignment to its dependencies. Let $\phi_{sk}$ denote the formula obtained from $\phi$ by replacing all existential variables $e$ by their Skolem function $f_e$. A DQBF $\psi = Q.\phi$ is satisfiable if and only if there exist Skolem functions $f_{e_1}, \ldots, f_{e_n}$, so that $\phi_{sk}$ is satisfied for all possible assignments to the universal variables of $\psi$.

*Universal expansion* is defined as the process of removing a universal variable $u$ from a formula $\psi$ considering both its values separately. This can be done by removing all existential variables $e$ with $u \in dep(e)$ and introducing two new existential variables $e_{u=1}, e_{u=0}$ with $dep(e_{u=1}) = dep(e_{u=0}) = dep(e)\backslash\{u\}$. Additionally, the matrix $\phi$ is replaced by $\phi_{u=1} \wedge \phi_{u=0}$. With $\phi_{u=v}$, we describe the formula obtained from $\phi$ by replacing $u$ by a constant $v \in \{1, 0\}$ and all occurrences of $e$ with $u \in dep(e)$ by $e_{u=v}$. We can use universal expansion to reduce any DQBF $\psi$ to an equisatisfiable propositional formula. If the resulting propositional formula is satisfiable, the Skolem functions of the original formula can be directly constructed from the assignments to the propositional variables by setting $f_e(v_1, \ldots, v_k) = e_{u_1=v_1,\ldots,u_k=v_k}$. In the following, we sometimes use the shorter notation $\phi_u$ and $\phi_{\overline{u}}$ instead of $\phi_{u=1}$ and $\phi_{u=0}$, respectively. We also extend this notation to clauses in the same way as we introduced it for formulas and refer to this as a *clause instance*, in the sense the Inst-Gen calculus [23, 24] uses *instantiation*, applied to the natural encoding of (D)QBF into first-order logic [35]. Furthermore, for a given clause instance $C_{l_1,\ldots,l_k}$, we define $ctx(C_{l_1,\ldots,l_k}) := \{l_i \mid i = 1, \ldots, k\}$. We call this the *context* of an instantiation.

The unique identifiers for the new existential variables introduced in this way make sure that the same existential variable is referred even if the individual clauses are considered separately. Also, the identifiers and the dependencies of all existential variables introduced during universal expansion are implicitly defined by the original quantifier prefix description. For example, for the DQBF

$$\forall u_1, u_2 \exists e_1(u_1), e_2(u_1, u_2), e_3(u_1, u_2) \;.\; (u_1 \vee e_1) \wedge (\overline{u}_2 \vee e_2) \wedge (\overline{u}_1 \vee u_2 \vee \overline{e}_3) \tag{1}$$

we can now write equations of clause instances such as:

$$\begin{aligned} &= (e_1)_{\overline{u}_1} \wedge (\overline{u}_2 \vee e_2) \wedge (\overline{u}_1 \vee u_2 \vee \overline{e}_3) &&= (e_1)_{\overline{u}_1} \wedge (e_2)_{u_2} \wedge (\overline{u}_1 \vee u_2 \vee \overline{e}_3) \\ &= (e_1)_{\overline{u}_1} \wedge (e_2)_{u_2} \wedge (u_2 \vee \overline{e}_3)_{u_1} &&= (e_1)_{\overline{u}_1} \wedge (e_2)_{u_2} \wedge (\overline{e}_3)_{u_1\overline{u}_2} \end{aligned}$$

The last line is a succinct representation of the full universal expansion of the original formula and minimal in the sense of our algorithm. We refer to each individual step as a *local universal expansion*. Note that we immediately dropped all trivially satisfied clauses (due to $u_i = 1$) in each step. Also, all intermediate steps can be performed in arbitrary order, e.g., although we started with expanding the first clause regarding $u_1$, it is not necessary to expand all other clauses on $u_1$ before expanding some clauses on $u_2$. Obviously, we could continue applying local universal expansion and obtain equivalent formulas of growing size:

$$(e_1)_{\overline{u}_1} \wedge (e_2)_{u_2} \wedge (\overline{e}_3)_{u_1\overline{u}_2} \quad = \quad (e_1)_{\overline{u}_1} \wedge (e_2)_{\overline{u}_1 u_2} \wedge (e_2)_{u_1 u_2} \wedge (\overline{e}_3)_{u_1\overline{u}_2}$$

The last expression is maximal and of the same size as the full universal expansion of $\psi$. There is no point in further expanding the first clause instance since $u_2 \notin dep(e_1)$, i.e. $(e_1)_{\overline{u}_1} = (e_1)_{\overline{u}_1\overline{u}_2} = (e_1)_{\overline{u}_1 u_2}$ Obviously, if a clause instance $C_{l_1,\ldots,l_k}$ is part of a formula, we can always add a more specific instance $C_{l_1,\ldots,l_k,l_{k+1},\ldots,l_{k'}}$ without affecting satisfiability. The more specific instance is actually subsumed by the original one, i.e. the full local universal expansion of the new instance is a subset of the full local universal expansion of the less specific one. This fact is crucial for the algorithm presented in Sect. 4.

EPR, known as the Bernays-Schönfinkel class, is a NExpTime-complete fragment of first-order logic [26]. It consists of the set of first-order formulas that, written in prenex form, contain (1) no function symbol of arity greater than 0, and (2) 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 $p(t_1, \ldots, t_n)$ where $p$ is a predicate symbol of arity $n$ and each $t_i$ is either a (universal) variable or a constant.

In [35], a translation from QBF to EPR is proposed. The approach consists of three steps and can be easily adapted to DQBF: (1) replace each existential variable $e$ with its Skolem function $f_e$ (which is in fact a predicate due to the Boolean domain), (2) replace each universal variable $u$ with $p(u)$ where $p$ is a fixed predicate, and (3) add the constraints $p(1)$ and $\neg p(0)$ to the formula. For example, for the DQBF in Eqn. (1) the resulting EPR formula is

$$\begin{aligned}\forall u_1, u_2 \;.\; &\big(p(u_1) \vee f_{e_1}(u_1)\big) \wedge \big(\neg p(u_2) \vee f_{e_2}(u_1, u_2)\big) \wedge \\ &\big(\neg p(u_1) \vee p(u_2) \vee \neg f_{e_3}(u_1, u_2)\big) \wedge p(1) \wedge \neg p(0)\end{aligned}$$

## 3 Related Work

The concepts of instantiation and expansion that we defined in Sect. 2 are similar to the notation used in [3], describing the solver sKizzo, which in particular shares similarities in the use of clause instances (c.f. *symbolic representation* in [3]). But apart from slightly different notation, there are three fundamental differences in the underlying algorithms: First, our method aims at solving DQBF while sKizzo, as described, targets QBF solving. Second, sKizzo uses a top-level QDPLL step, which cannot be applied to DQBF formulas without introducing additional concepts as presented in our previous work [14]. Finally, the most important difference is that sKizzo performs a full Skolemization after preprocessing, while our solver uses local extension to iteratively generate a (potentially exponentially) more succinct formula which is sufficient to prove (un)satisfiability of the original input, as described in Sect. 4.

Another similar notation and related work is proposed in [19, 20, 21]. Their solver RAReQS [20] creates propositional abstractions and uses a CEGAR approach [10] for refinement. As we will discuss in Sect. 4, this is also what our solver does. However, the way abstractions are generated and refined is different. One main difference can be found in the expansion of universal variables. In contrast to sKizzo, both, RAReQS as well as our solver, allow partial expansion in the sense that only $\phi_{u=1}$ or $\phi_{u=0}$ might be considered for some formula $\phi$ containing $u$. Nevertheless, even the restricted expansion of universal variables in [19, 20, 21] always applies to *all* clauses of a formula, whereas our approach uses the previously described concept of *local universal expansion*, which allows to expand clauses individually. Further, RAReQS is a QBF solver and cannot tackle DQBF formulas. Due to the usage of recursive

```
$F' :=$ initInstantiation$(F)$
while $true$ do
    $F'' =$ propositionalAbstraction$(F')$
    $(state, assignment) =$ checkSat$(F'')$
    if $(state == unsat)$ then return $unsat$
    if isValid$(assignment, F, F')$ then return $sat$
    $F' =$ refineInstantiation$(assignment, F, F')$
```

Figure 1: Pseudo-code of a CEGAR loop as used in the Inst-Gen procedure [22, 23, 24].

calls depending on the order of the quantifier prefix, an extension to DQBF does not seem to be straightforward.

Another solver that relies on abstraction refinement, is given in [38]. While they target quantified bit-vector formulas with uninterpeted functions, QBF and DQBF of course can be seen as a special case. To generate abstractions, they apply Skolemization and use templates for functions. The effectiveness of their approach heavily relies on the right choice of templates, which can be difficult for QBF and DQBF. Finally, another algorithm that has a similar structure can be found in [34]. Again, their solver actually targets more general SMT formulas, but could theoretically also be used for QBF. Since their approach expects an ordered quantifier prefix, it cannot be directly applied to DQBF.

## 4 iDQ architecture

In this section, we present the iDQ architecture. It is based on the more general Inst-Gen calculus [23, 24] for EPR as used in iProver [22], but reduced to the more specific case of DQBF. Instead of dealing with predicates, we use the notion of clause instances as introduced in Sect. 2. The Inst-Gen architecture is based on the CEGAR paradigm [10] and the pseudo-code is given in Fig. 1.

For EPR, usually no specific initial instantiation is used, i.e., the formula is completely uninstantiated. A propositional abstraction is then created by grounding the current formula and can be solved by a SAT solver. If the SAT solver returns *unsat*, the original formula is *unsat* too, since the ground formula is an overapproximation. On the other hand, if the SAT solver returns *sat*, the resulting assignment has to be checked for consistency with the EPR formula. In each propositional clause, we select a satisfying literal, determined by a fixed *selection function*. If there is no pair of oppositely signed, selected literals, such that the corresponding EPR literals can be unified, the solution is also valid for the original EPR formula. If there are such pairs of literals, then we try to apply the following *inference step* to each corresponding EPR clause: apply the most general unifier (MGU) to the clause and add the result as a new clause. By checking if the new clause is already part of the formula w.r.t. some *redundancy* concept, it is also possible that no new clause is added. The formula is then called *saturated* and the current assignment is also valid for the input formula. Otherwise, the calculus starts the next iteration.

Using the approach described in [35], any DQBF can be translated to EPR. All universal variables $u$ are embedded into EPR by introducing a predicate $p$ and replacing each occurrence of $u$ by $p(u)$. Additionally, the constraints $p(1)$ and $\neg p(0)$ are added to the formula. Obviously, this implies that $p(u)$ and $\neg p(u)$ can never end up being the only satisfying literal of a clause. If this was the case, unification with $p(1)$ and $\neg p(0)$ would be possible, respectively. As a result, the corresponding instance would be added to the formula and, from that point on, in every loop iteration the SAT solver would immediately set the instantiated literal to 0 by unit propagation.

Knowing that we deal with DQBF, this will always be the case. Therefore, we can directly simplify the formula in the beginning by starting with a more specific initial instantiation. For each clause, we only care about those assignments to the universal variables which do not trivially satisfy the clause. In our notation, this initial instantiation is equal to the minimal instantiation created by local universal expansion as described in Sect. 2. Consider the following example:

$$\psi \;=\; \forall u_1, u_2 \exists e_1(u_1, u_2), e_2(u_2)\ .\ (u_1 \vee e_1) \wedge (u_1 \vee \overline{e}_1) \wedge (\overline{u}_1 \vee u_2 \vee e_1 \vee e_2)$$

We now create the initial set of clause instances, using the unique minimal instantiation that removes all universal variables from the clauses:

$$(e_1)_{\overline{u}_1} \wedge (\overline{e}_1)_{\overline{u}_1} \wedge (e_1 \vee e_2)_{u_1\overline{u}_2}$$

We then create a propositional abstraction of the current clause instance set, by assuming that all existential variables that do not occur in the same instantiation context can be different. This means, for $\mathfrak{P}$ denoting the power-set, we use a function $m : E \times \mathfrak{P}(\{l \mid var(l) \in U\}) \rightarrow V'$ for some new set of propositional variables $V'$, and map each literal $e$ in a clause instance $C$ to a propositional variable $m(e, ctx(C))$. We restrict $m$ as follows:

$$m(e_1, ctx(C_1)) = m(e_2, ctx(C_2)) \qquad \text{if and only if}$$
$$e_1 = e_2,\ \big\{l_1 \in ctx(C_1) \mid var(l_1) \in dep(e_1)\big\} = \big\{l_2 \in ctx(C_2) \mid var(l_2) \in dep(e_2)\big\}$$

Obviously, the propositional formula generated by this mapping is an overapproximation of the current set of clause instances. It will often be the case that there is some kind of dependency between different variables.

In our example, we get the following propositional formula:

$$(x_1) \wedge (\overline{x}_1) \wedge (x_2 \vee x_3)$$

Satisfiability can easily be checked by using any off-the-shelf SAT solver. In this specific example, the propositional overapproximation is unsatisfiable. This implies that the original formula is also unsatisfiable.

If, on the other hand, the propositional formula was satisfiable, we would need additional reasoning. For this, consider a second example:

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

Again, we create the initial set of clause instances using the unique minimal instantiation that removes all universal variables from the clauses:

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

The propositional overapproximation now looks as follows:

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

Note that the same existential variable $e_1$ is mapped to two different variables $x_1, x_2$ because it appears in different contexts. The SAT solver would now tell us that this abstraction is satisfiable and return a satisfying assignment $\alpha$, e.g., $\alpha = \{x_1 \rightarrow 1, x_2 \rightarrow 0, x_3 \rightarrow 0\}$.

We now check, whether $\alpha$ is a valid satisfying assignment for the current set of clause instances. This is the case if and only if no pair of oppositely signed, selected (satisfying) literals corresponds to the same existential variable in overlapping contexts. For EPR, this is exactly what happens in the Inst-Gen calculus when there is a check on whether the corresponding literals can be unified [22, 23, 24]. In the case that a satisfying assignment is valid for the current set of clause instances, we know that the original DQBF is satisfiable. If, however, the assignment is not valid, we refine the instantiation on the clauses that contain the conflicting literals by adding new instances. Those instances are actually subsumed by the original ones but lead to a different propositional abstraction by the definition of $m$. In the next step, the propositional abstraction will automatically rule out this conflicting assignment.

In our latest example, $\alpha$ is indeed not a valid assignment for the current set of clause instances: $x_1$ and $x_2$ correspond to $e_1$, appear in overlapping contexts and, therefore, the propositional variables cannot be assumed to be independent of each other. We therefore apply the inference step of *merging* the two contexts and adding new clause instances. Now, the resulting formula looks 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 propositional abstraction is given by:

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

Note that $e_2$ is mapped to the same variable $x_4$ in both clause instances although it appears in a different instantiation context. This is due to $u_1 \notin \mathit{dep}(e_2)$, which implies that $(e_2)_{u_2} = (e_2)_{\overline{u}_1 u_2}$. Again, this propositional formula is satisfiable and the SAT solver could return a satisfying assignment $\alpha = \{x_1 \to 1, x_2 \to 1, x_3 \to 0, x_4 \to 1\}$. However, this time we can pick a literal in each clause so that no implicit dependencies are violated. Therefore, the algorithm terminates and the original formula is known to be satisfiable.

Furthermore, also note that in our particular case, we could have directly applied local universal expansion to our instances instead of adding a single more specific one, e.g., yielding $(e_1)_{\overline{u}_1\overline{u}_2} \wedge (e_1)_{\overline{u}_1 u_2}$ instead of $(e_1)_{\overline{u}_1} \wedge (e_1)_{\overline{u}_1 u_2}$. However, this can only be done without growth in formula size, if there is exactly one additional literal in the new context of the instance, which we would have added otherwise. Nevertheless, this is a possible DQBF-specific extension, which is part of future work, and sometimes might reduce the number of loop iterations.

## 5 Implementation

In this section, we describe how we actually implemented the proposed algorithm and point out where we can profit from DQBF-specific restrictions. For our solver, we use input files in a format that is an extension of the `QDIMACS` format and which we call `DQDIMACS`. The only difference to `QDIMACS` is the fact that we additionally allow partially ordered dependencies by using expressions of the form `d <int32> [<int32> ... <int32>] 0` in the quantifier prefix description. This defines a new existential variable given by the first ID as integer which (optionally) depends on a list of previously defined universal variables. All other quantifier definitions using `a` and `e` are still interpreted in the same way as it is done in the `QDIMACS` format and existential variables defined by using `e` are assumed to depend on all previously defined universal variables as usual. In this way, `DQDIMACS` is easy to parse and a real extension of `QDIMACS`. `DQDIMACS` is also the input format which we use in all our experiments in Sect. 6.

Universal variables: $u_1, u_2, u_3$
Existential variable: $e(u_1, u_3) \longrightarrow$ (D) dependency mask: 101
Input clause: $(u_2 \vee \overline{u}_3 \vee e)$
(I) Initial instance: $(e)_{\overline{u}_2 u_3} \longrightarrow$ 011 / 001
(C) $e$'s concrete context: $u_3 \longrightarrow$ D & I1 = 001 / C1 & I2 = 001
(G) $e$'s ground context: $\overline{u}_1 u_3 \longrightarrow$ D = 101 / I2 = 001

(a) Dependencies and contexts.

(A) Instance with $e$: $(e \vee \dots)_{\overline{u}_1 u_2 u_3} \rightarrow$ 111 / 011
(B) Instance with $\overline{e}$: $(\overline{e} \vee \dots)_{\overline{u}_2 u_3} \longrightarrow$ 011 / 001
(E) $e$'s concrete context: $\overline{u}_1 u_3 \longrightarrow$ 101 / 001
(F) $\overline{e}$'s concrete context: $u_3 \longrightarrow$ 001 / 001
Overlapping contexts? $\longrightarrow$ (E1 & F1) & E2 $\overset{?}{=}$ (E1 & F1) & F2
(N) New instance: $(\overline{e} \vee \dots)_{\overline{u}_1 \overline{u}_2 u_3} \longrightarrow$ B1 | E1 = 111 / B2 | E2 = 001
Redundant? $\longrightarrow$ (1) ground B $\overset{?}{=}$ ground N (both are 111 / 001)
(2) B1 & N1 $\overset{?}{=}$ B1 and B1 & N2 $\overset{?}{=}$ B1 & B2

(b) Inference step and redundancy check.

Figure 2: Examples of using bit-vector representation for various calculations in iDQ.

After parsing the input, the data structures we use are similar to those of common SAT solvers. The matrix of the original formula is saved as a list of clauses and a clause is saved as a list of literals represented by integers. Additionally, the quantifier prefix is saved as a list of variables and each variable has an ID, a quantifier type and, if it is an existential variable, a bit-vector, called the *dependency mask*, representing the universal variables that it depends on.

We store a *list of instances* with each clause. An instance is defined by two bit-vectors, called the *context mask* and the *value mask*, representing the universal variables that are assigned by the context and the values they are assigned to, respectively. E.g., see instance (I) in Fig. 2(a), where the first mask is the context mask and the second one is the value mask. For the propositional abstraction, a *propositional clause* is also stored with each instance. All propositional clauses are incrementally added to the underlying SAT solver, PicoSAT [5].

**Initial Instantiation.** Creating the initial instantiation is straightforward. When parsing the clauses of the formula, universal literals $l$ are not added to the literal list of the current clause, but instead the corresponding bits in the context mask and the value mask are set accordingly to represent that $\overline{l}$ is part of the context of the current instance; see (I) in Fig. 2(a).

**Propositional Abstraction.** Each occurrence of each existential variable is mapped to a corresponding propositional variable. This can be done efficiently by using the bit-vectors that are saved with each existential variable and each clause instance. Given an existential variable $e$ that occurs in an instance $c$, we calculate $e$'s *concrete context* that is to show which part of $c$'s context is relevant for $e$. The concrete context can be calculated by applying *bitwise and* to $e$'s dependency mask and $c$'s context mask, and another *bitwise and* with $c$'s value mask. This is illustrated in Fig. 2(a) as the context mask (C1) being calculated from (D) and (I1), and the value mask (C2) from (C1) and (I2).

We map the variable ID and the concrete context to a unique propositional variable. Accordingly, if two variable occurrences have the same ID (i.e., they represent the same existential variable) and their concrete contexts are equal, they are mapped to the same propositional variable. In order to check whether we already introduced the corresponding propositional variable in a previous step, we keep a hash table with all previously introduced propositional variables.

**Grounding.** As the Inst-Gen calculus [23, 24] suggests, before mapping an existential variable $e$ and its concrete context to a propositional variable, iDQ generates the grounding of this context. Grounding is basically about assigning a concrete truth value, w.l.o.g. 0, to all the universal variables which $e$ depends on and which are not already assigned by the context. This can easily be done by setting the context mask to $e$'s dependency mask and leaving the value mask as it is, assuming that all bits in our bit-vectors are initialized to 0. Fig. 2(a) shows an example, as setting (G1) to (D) and (G2) to (I2).

**Active and Passive Instances.** Similar to iProver's architecture, clause instances are separated into two sets, called *active* and *passive*. Active instances are the ones among which all possible inference steps have been performed, modulo literal selection. Passive instances are the ones which are waiting to participate in inferences. In iDQ, passive instances are stored in a *priority queue* ordered by a given heuristic. In each solving iteration, iDQ dequeues a given number of passive instances with the highest priority, and sets them active one by one, which involves trying to apply an inference step with each active instance.

In the current implementation of iDQ, an active instance does not move back to the passive instance set whenever its literal selection changes, as opposed to iProver. We rather apply inference steps to it with each active instance, on the newly selected literal.

An inference step on two selected literals can easily be implemented, as illustrated in Fig. 2(b). First, to check whether the concrete contexts of the literals are overlapping, we apply *bitwise and*. Second, to calculate the context and value masks for a new instance, we apply *bitwise or* to the masks representing the original instance and the ones representing the literal from the other instance.

**Heuristics.** Two choices depend on some heuristics: (1) how to order the priority queue of passive instances, and (2) how to select a satisfying literal in an active instance. We have been experimenting with two types of heuristics, using different criteria for both choices.

One of the heuristics is inspired by iProver's default heuristic, based on the lexicographical combination of orders defined on given numerical/Boolean parameters. Similar to iProver's notation [24], we use the following combinations: (1) `[-num_dep;+age;-num_symb]` for the priority queue of instances, and (2) `[+sign;+ground;-num_dep;-num_symb]` for literal selection. I.e., priority is given to instances with fewer unassigned dependencies, then to instances generated at earlier iterations, and finally to instances with fewer symbols (0 or 1) assigned to dependencies. The heuristic for literal selection can be interpreted in a similar way, where positive and then ground literals are prioritized the most.

The other heuristic is inspired by SAT solving. It is based on the VSIDS scores [29] of propositional variables used in the propositional abstraction. iDQ counts the occurrences of those variables in the propositional clauses generated so far, and then, after each 50 iterations, all the scores are divided by 2. Based on the VSIDS scores, (1) priority is given to the passive instance with the highest average score of its literals, and (2) the literal with the highest score is selected.

**Redundancy Check.** Redundancy elimination is crucial for the applicability of any calculus, in order to avoid infinite runs and to obtain a smaller knowledge base. Due to the finite domain property, it is easy to obtain a sufficient, but not practical, redundancy check for both EPR and DQBF, by simply checking the equality of clause instances, i.e., of context/value masks in iDQ.

However, a practical redundancy check might be more complicated, e.g., iProver employs dismatching constraints [24]. With iDQ, a practical check can be obtained more easily. iDQ decides if a new instance $c$ would not give any new information to the active instance set, meaning that the propositional abstraction would stay the same and all inference steps with $c$ would also result in redundant instances. We consider $c$ redundant if there exists an active instance $d$ of the *same* clause such that (1) the propositional abstractions of $c$ and $d$ are the same, and (2) $d$ subsumes $c$. Both checks can be done by bit-vector operations, as illustrated in Fig. 2(b). Importantly, (2) requires to check if $c$'s context is a superset of $d$'s contexts.

## 6 Experimental Results

In this section, we report experiments[1] with our solver. The source code, benchmarks, and log files are available at `http://fmv.jku.at/idq`. We tested iDQ with two types of heuristics as proposed in Sect. 5. iDQ and iDQ$_{\text{vsids}}$ refer to the versions that employ the default heuristic and the VSIDS-based heuristic, respectively. Lacking in publicly available, general-purpose DQBF solvers (the solver DQBF2QBF in [13] can reason only with *unsat* formulas), we decided to also compare iDQ against iProver (v0.8.1).

We also tested iDQ on QBF benchmarks, by exploiting the fact that QBF is a real fragment of DQBF. By doing so, we could compare iDQ not only against iProver, but also against genuine QBF solvers, like the QDPLL-based DepQBF [28] (v3.0), the Skolemization-based sKizzo [3] (v0.8.2), the CEGAR-based RAReQS [20] (v1.1), and the expansion-based Nenofex [27] (v1.0). For the sake of fair comparison, we did not run any preprocessor.

**DQBF Benchmarks.** We used the only publicly available DQBF benchmarks by Finkbeiner and Tentrup [13]. All of them encode partial equivalence checking (PEC) problems, i.e., circuits containing some "black boxes" compared against full circuits. This benchmark set includes the benchmarks of the 3-bit arithmetic circuits *adder* and the 16-bit arbiter implementations *bitcell* and *lookahead* from [11], and also the circuit family *pec_ xor* from [16] about comparing the XOR of input bits against a random Boolean function. We converted those benchmarks to `DQDIMACS` format, and then ran iDQ on them. For iProver, we further converted the `DQDIMACS` instances to EPR (TPTP CNF format) by using the translation from [35], which can be easily adapted to DQBF.

Table 1 shows the results: the number of solved instances (#), the number of timeouts (TO), and the average runtime. The number at the end of benchmark names shows the number of black boxes in circuits. In most of the cases, iDQ outperforms iProver. iDQ$_{\text{vsids}}$ performs even better than iDQ on the *bitcell* benchmarks but worse on the *lookahead* and *adder* benchmarks. The gap between the performance of iDQ and iProver is significant. On *unsat* instances, DQBF2QBF generally is the fastest solver. However, the performance of iDQ sometimes comes quite close, whereas DQBF2QBF cannot solve *sat* instances at all. Also note that the benchmarks are biased in the way that most sets contain mainly *unsat* instances. Finally, we think that one reason for the better performance of DQBF2QBF on *unsat* instances is the better encoding of the original benchmarks and the overhead introduced by CNF translation.

[1]Setup: Vienna Scientific Cluster (VSC-2), AMD Opteron Magny Cours 6132HE CPUs, 2.2 GHz cores, 900 seconds time limit, 3800 MB memory limit.

Preliminary results on simple preprocessing techniques show that this can lift the performance of iDQ to come even closer to the one of DQBF2QBF.

| | *#(sat/uns)* | *TO* | *time* | *#(sat/uns)* | *TO* | *time* | *#(sat/uns)* | *TO* | *time* |
|---|---|---|---|---|---|---|---|---|---|
| | bitcell_16_2 | | | bitcell_16_4 | | | bitcell_16_6 | | |
| DQBF2QBF | 98 (0/98) | 2 | 18.6 | 98 (0/98) | 2 | 18.8 | 97 (0/97) | 3 | 27.8 |
| iDQ | 88 (2/86) | 12 | 128.1 | 52 (0/52) | 48 | 488.9 | 22 (0/22) | 78 | 735.9 |
| iDQ$_{\text{vsids}}$ | 97 (2/95) | 3 | 39.2 | 75 (0/75) | 25 | 255.9 | 36 (0/36) | 64 | 592.0 |
| iProver | 82 (0/82) | 18 | 248.6 | 34 (0/34) | 66 | 684.5 | 7 (0/7) | 93 | 851.7 |
| | lookahead_16_2 | | | lookahead_16_4 | | | lookahead_16_6 | | |
| DQBF2QBF | 97 (0/97) | 3 | 27.7 | 97 (0/97) | 3 | 27.7 | 96 (0/96) | 4 | 36.6 |
| iDQ | 98 (3/95) | 2 | 30.4 | 88 (0/88) | 12 | 118.9 | 69 (0/69) | 31 | 342.4 |
| iDQ$_{\text{vsids}}$ | 93 (2/91) | 7 | 68.1 | 62 (0/62) | 38 | 383.0 | 20 (0/20) | 80 | 729.9 |
| iProver | 67 (0/67) | 33 | 351.8 | 32 (0/32) | 68 | 656.3 | 6 (0/6) | 94 | 862.9 |
| | adder_3_2 | | | adder_3_4 | | | adder_3_6 | | |
| DQBF2QBF | 94 (0/94) | 6 | 54.8 | 89 (0/89) | 11 | 99.8 | 74 (0/74) | 26 | 234.6 |
| iDQ | 82 (1/81) | 18 | 246.8 | 58 (0/58) | 42 | 440.2 | 11 (0/11) | 89 | 841.4 |
| iDQ$_{\text{vsids}}$ | 43 (0/43) | 57 | 546.3 | 21 (0/21) | 79 | 734.0 | 6 (0/6) | 94 | 863.9 |
| iProver | 86 (1/85) | 14 | 221.6 | 54 (0/54) | 46 | 538.2 | 5 (0/5) | 95 | 876.9 |
| | pec_xor2 | | | pec_xor3 | | | pec_xor4 | | |
| DQBF2QBF | 49 (0/49) | 51 | 459.4 | 77 (0/77) | 23 | 207.5 | 99 (0/99) | 1 | 10.6 |
| iDQ | 100 (51/49) | | .5 | 100 (23/77) | | .7 | 100 (1/99) | | 3.3 |
| iDQ$_{\text{vsids}}$ | 100 (51/49) | | .5 | 100 (23/77) | | .6 | 100 (1/99) | | 2.2 |
| iProver | 100 (51/49) | | .5 | 100 (23/77) | | .9 | 100 (1/99) | | 2.8 |

Table 1: Results for *DQBF PEC* benchmarks

**QBF Benchmarks.** We used *QBF Gallery 2013* benchmarks, from which we selected instances whose size do not exceed 2 megabytes. In some cases, we randomly selected instances from the resulting sets. Table 2 shows the results, including the number of memory outs (MO) and the number of crashes (CR). Between parentheses after each benchmark name, the number of instances is shown. As expected, genuine QBF solvers outperform iDQ and iProver on most benchmarks, although sKizzo and Nenofex terminate with memory out quite frequently. On some instances, iProver and Nenofex crash. iDQ performs particularly well on the benchmarks *conformant-planning* and *planning-CTE*, and reasonably well on *sauer-reimer*. In general, the VSIDS-heuristic seems to be the slightly better choice.

## 7 Conclusion

In this paper, we presented an instantiation-based algorithm for solving DQBF, resulting in a complete and at the same time practical DQBF solver.

On the theoretic side, we showed how successful techniques in EPR solving can be lifted to the more specific DQBF case. We brought together related work on Skolemization with the Inst-Gen calculus. On the other hand, we extended work on iProver by giving a simpler framework. While our implementation is still a prototype, our experiments confirmed that the

| | *#(sat/uns)* | *TO/MO* | *time* | *CR* | *#(sat/uns)* | *TO/MO* | *time* | *CR* |
|---|---|---|---|---|---|---|---|---|
| | conformant-planning (100) | | | | planning-CTE (57) | | | |
| DepQBF | 89 (19/70) | 11/0 | 130.7 | | 42 (26/16) | 15/0 | 297.0 | |
| RAReQS | 94 (17/77) | 4/2 | 49.1 | | 57 (35/22) | | 1.4 | |
| Nenofex | 95 (19/76) | | 19.7 | 5 | 57 (35/22) | | 3.8 | |
| sKizzo | 51 (11/40) | 34/15 | 380.9 | | 57 (35/22) | | 1.8 | |
| iDQ | 95 (14/81) | 5/0 | 81.9 | | 57 (35/22) | | 6.2 | |
| iDQ$_{\text{vsids}}$ | 95 (14/81) | 5/0 | 80.2 | | 57 (35/22) | | 6.5 | |
| iProver | 91 (14/77) | 9/0 | 90.9 | | 57 (35/22) | | 4.6 | |
| | qbf-hardness (162) | | | | reduction-finding (100) | | | |
| DepQBF | 59 (12/47) | 103/0 | 586.1 | | 65 (34/31) | 35/0 | 348.4 | |
| RAReQS | 63 (12/51) | 99/0 | 572.0 | | 81 (41/40) | 19/0 | 201.2 | |
| Nenofex | 26 (12/14) | 0/136 | 487.9 | | 35 (19/16) | 0/65 | 425.0 | |
| sKizzo | 48 (12/36) | 79/35 | 526.8 | | 34 (19/15) | 46/20 | 468.2 | |
| iDQ | 44 (12/32) | 118/0 | 665.0 | | 30 (16/14) | 70/0 | 635.4 | |
| iDQ$_{\text{vsids}}$ | 42 (12/30) | 120/0 | 666.8 | | 29 (15/14) | 64/7 | 598.2 | |
| iProver | 26 (12/14) | 135/0 | 762.8 | 1 | 31 (18/13) | 48/6 | 554.9 | 15 |
| | sauer-reimer (100) | | | | eval2012r2 (264) | | | |
| DepQBF | 50 (35/15) | 50/0 | 457.3 | | 90 (33/57) | 174/0 | 610.8 | |
| RAReQS | 33 (20/13) | 0/67 | 248.2 | | 67 (23/44) | 162/35 | 626.7 | |
| Nenofex | 18 (9/9) | 0/82 | 564.7 | | 54 (28/26) | 7/200 | 519.0 | 3 |
| sKizzo | 18 (9/9) | 43/39 | 614.8 | | 89 (39/50) | 128/47 | 521.7 | |
| iDQ | 20 (8/12) | 80/0 | 724.7 | | 45 (15/30) | 217/2 | 757.8 | |
| iDQ$_{\text{vsids}}$ | 27 (17/10) | 73/0 | 658.7 | | 51 (18/33) | 178/35 | 682.2 | |
| iProver | 19 (10/9) | 76/5 | 725.8 | | 54 (18/36) | 178/30 | 672.7 | 2 |

Table 2: Results for *QBF Gallery 2013* benchmarks

simpler structure of DQBF compared to the more general EPR, as well as the smaller formula size compared to the full expansion, can have a positive impact on solver performance.

So far, our optimization compared to iProver was mainly on the implementation side using more efficient data structures and operations tailored to the Boolean domain. Apart from the possibility of applying local universal expansion as a special case of instantiation, looking into more potential DQBF-specific benefits, especially on the heuristical level, is part of future work. Specialized preprocessing techniques, e.g., related to those applied in sKizzo [3] or for general QBF solvers [6], as well as removing dependencies of existential variables by analyzing the propositional matrix [28], might also be a further interesting step into the direction of even more efficient DQBF solving.

Another potential benefit of our solver could be related to providing certificates. Certificate construction in QBF has seen increasing interest in recent research [8, 12, 18, 19, 21, 30, 36, 37]. While providing certificates is not implemented in our prototype yet, our architecture can easily be extended by this feature. Obviously, Skolem functions for satisfying formulas can directly be constructed out of a solution as discussed in Sect. 2. However, the more interesting contribution might be for unsatisfiable formulas. As unsatisfiability of a formula is proven by a SAT solver in combination with universal expansion, we can directly use the generated resolution proof for refuting the initial DQBF input, similar to the approach described in [21]. Due to the iterative refinement in the solving process, certificates (for unsatisfiability as well as satisfiability) might be rather small. Further shrinking could be possible by looking for unsatisfiable cores.

Finally, we were able to outperform even more specific QBF solvers on some benchmarks. As an additional side-effect, we therefore hope to get new insights into QBF solving and maybe even QBF solvers might profit from our techniques.

## References

[1] Valeriy Balabanov, Hui-Ju Katherine Chiang, and Jie-Hong R. Jiang. Henkin quantifiers and boolean formulae. In *Proc. SAT'12*, 2012.

[2] Valeriy Balabanov, Hui-Ju Katherine Chiang, and Jie-Hong R Jiang. Henkin quantifiers and boolean formulae: A certification perspective of DQBF. *Theoretical Computer Science*, 2013.

[3] Marco Benedetti. Evaluating QBFs via symbolic skolemization. In *Proc. LPAR'04*, pages 285–300, 2004.

[4] Armin Biere. Resolve and expand. In *Proc. SAT'04*, pages 238–246, 2004.

[5] Armin Biere. PicoSAT essentials. *JSAT*, 4(2-4):75–97, 2008.

[6] Armin Biere, Florian Lonsing, and Martina Seidl. Blocked clause elimination for QBF. In *Proc. CADE'11*, pages 101–115, 2011.

[7] U. Bubeck and H. Kleine Büning. Dependency quantified horn formulas: Models and complexity. In *Proc. SAT'06*, 2006.

[8] Uwe Bubeck and Hans Kleine Büning. Nested boolean functions as models for quantified boolean formulas. In *Proc. SAT'13*, pages 267–275, 2013.

[9] Krishnendu Chatterjee, Thomas A Henzinger, Jan Otop, and Andreas Pavlogiannis. Distributed synthesis for LTL fragments. In *Proc. FMCAD'13*, pages 18–25, 2013.

[10] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, September 2003.

[11] William J. Dally and R. Curtis Harting. *Digital Design, A Systems Approach.* Cambridge University Press, 2012.

[12] Uwe Egly, Florian Lonsing, and Magdalena Widl. Long-distance resolution: Proof generation and strategy extraction in search-based qbf solving. In *Proc. LPAR'13*, pages 291–308, 2013.

[13] Bernd Finkbeiner and Leander Tentrup. Fast DQBF refutation. In *Proc. SAT'14*, 2014.

[14] Andreas Fröhlich, Gergely Kovásznai, and Armin Biere. A DPLL algorithm for solving DQBF. In *Proc. POS'12*, 2012.

[15] Karina Gitina, Sven Reimer, Matthias Sauer, Ralf Wimmer, Christoph Scholl, and Bernd Becker. Equivalence checking for partial implementations revisited. In *Proc. MBMV'13*, pages 61–70, 2013.

[16] Karina Gitina, Sven Reimer, Matthias Sauer, Ralf Wimmer, Christoph Scholl, and Bernd Becker. Equivalence checking of partial designs using dependency quantified boolean formulae. In *Proc. ICCD'13*, pages 396–403, 2013.

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

[18] Marijn J. H. Heule, Martina Seidl, and Armin Biere. A Unified Proof System for QBF Preprocessing. In *Proc. IJCAR'14*, 2014.

[19] Mikoláš Janota, Radu Grigore, and Joao Marques-Silva. On QBF proofs and preprocessing. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 473–489, 2013.

[20] Mikoláš Janota, William Klieber, Joao Marques-Silva, and Edmund Clarke. Solving QBF with counterexample guided refinement. In *Proc. SAT'12*, pages 114–128, 2012.

[21] Mikoláš Janota and Joao Marques-Silva. On propositional QBF expansions and Q-resolution. In *Proc. SAT'13*, pages 67–82, 2013.

[22] Konstantin Korovin. iProver – an instantiation-based theorem prover for first-order logic (system description). In *Proc. IJCAR'08*, 2008.

[23] Konstantin Korovin. Instantiation-based automated reasoning: From theory to practice. In *Proc. CADE'09*, pages 163–166, 2009.

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

[25] Gergely Kovásznai, Andreas Fröhlich, and Armin Biere. On the complexity of fixed-size bit-vector logics with binary encoded bit-width. In *Proc. SMT'12*, 2012.

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

[27] Florian Lonsing and Armin Biere. Nenofex: Expanding NNF for QBF solving. In *Proc. SAT'08*, pages 196–210, 2008.

[28] Florian Lonsing and Armin Biere. Integrating dependency schemes in search-based QBF solvers. In *Proc. SAT'10*, pages 158–171, 2010.

[29] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proc. DAC'01*, pages 530–535, 2001.

[30] Aina Niemetz, Mathias Preiner, Florian Lonsing, Martina Seidl, and Armin Biere. Resolution-based certificate extraction for QBF (tool presentation). In *Proc. SAT'12*, 2012.

[31] Christos H. Papadimitriou. *Computational complexity*. Addison-Wesley, 1994.

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

[33] Gary L. Peterson and John H. Reif. Multiple-person alternation. In *Proc. FOCS'79*, pages 348–363, 1979.

[34] Anh-Dung Phan, Nikolaj Bjørner, and David Monniaux. Anatomy of alternating quantifier satisfiability (work in progress). In *Proc. SMT'12*, pages 120–130, 2012.

[35] Martina Seidl, Florian Lonsing, and Armin Biere. QBF2EPR: A tool for generating EPR formulas from QBF. In *Proc. PAAR'12*, 2012.

[36] Friedrich Slivovsky and Stefan Szeider. Variable dependencies and Q-resolution. In *International Workshop on Quantified Boolean Formulas 2013, Informal Workshop Report*, page 22, 2013.

[37] Friedrich Slivovsky and Stefan Szeider. Variable Dependencies and Q-Resolution. In *Proc. SAT'14*, 2014.

[38] Christoph M. Wintersteiger, Youssef Hamadi, and Leonardo de Moura. Efficiently solving quantified bit-vector formulas. In *Proc. FMCAD'10*, 2010.