Preprint from Proceedings of Design, Automation & Test in Europe Conference (DATE)
March 2015, Grenoble, France

Solving DQBF Through Quantifier Elimination*

Karina Gitina, Ralf Wimmer, Sven Reimer, Matthias Sauer, Christoph Scholl, and Bernd Becker
Institute of Computer Science, Albert-Ludwigs-Universität Freiburg
Georges-Köhler-Allee 051, D-79110 Freiburg, Germany
{gitina | wimmer | reimer | sauerm | scholl | becker}@informatik.uni-freiburg.de

***Abstract*—We show how to solve dependency quantified Boolean formulas (DQBF) using a quantifier elimination strategy which yields an equivalent QBF that can be decided using any standard QBF solver. The elimination is accompanied by a number of optimizations which help reduce memory consumption and computation time.**

We apply our solver HQS to problems from the domain of verification of incomplete combinational circuits to demonstrate the effectiveness of the proposed algorithm. The results show enormous improvements both in the number of solved instances and in the computation times compared to existing work on validating DQBF.

I. Introduction

The last two decades have brought enormous progress in the solution of quantifier-free Boolean formulas—the famous NP-complete SAT problem. While first solvers for SAT were developed in the early 1960s, the actual breakthrough began in the mid-1990s with techniques like conflict-driven clause learning, non-chronological backtracking, watched-literal schemes and many more [1]. They reduced the computational effort to decide the satisfiability of a given formula by many orders of magnitude. Nowadays, formulas with millions of clauses and hundred thousands of variables can be solved.

These algorithmic advances paved the way to a successful adoption of SAT-based techniques in a wide range of applications, among others: hard- and software verification (e. g., for bounded model checking [2], [3]), test pattern generation [4], [5], and planning [6].

At the same time it was realized that other applications are hard to model using quantifier-free formulas, but are expressible in a more natural and compact way using quantified Boolean formulas (QBF). Many lessons learned from the development of efficient SAT-solvers could be transferred to the quantified case. Together with further improvements specific to QBF, QBF solvers are becoming a serious possibility to tackle computationally hard problems [7].

The success of SAT- and QBF-based methods encourages us to investigate even more complex Boolean decision problems. Deciding so-called dependency quantified Boolean formulas (DQBF), which are a generalization of QBF, is NEXPTIME-complete [8]. In constrast to QBF, where variable dependencies always follow a linear order (each existential variable depends on all universal variables left of its position in the quantifier prefix), the dependencies in DQBF are explicitly stated by Henkin-quantifiers [9] allowing also non-linear dependencies which are not expressible with QBF. For many relevant problems from this complexity class there are natural transformations into equivalent DQBF problems. Examples are realizability of incomplete digital circuits [10], the analysis of non-cooperative games with incomplete information [8], and certain bit-vector logics [11], [12]. We expect—as it was the case with SAT and QBF—even more applications to come with the availability of efficient DQBF solvers.

*This work was partly supported by the German Research Council (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS).

Currently the lack of efficient DQBF solvers severely limits its applicability to practical problems. While Balabanov et al. [13] investigate the theory of DQBF, a first extension of the QDPLL algorithm for deciding QBF to DQBF was proposed by Fröhlich et al. [14], but without experimental evaluation. Finkbeiner and Tentrup [15] gave an incomplete algorithm for the refutation of DQBF, which is not able to give conclusive answers on satisfied instances. A basic variable elimination strategy is presented in [10] demonstrating the need for DQBF formulations experimentally on a series of small incomplete circuits. The DQBF formulations used in [10] solve the so-called realizability problem for incomplete digital designs, i. e., they investigate the question whether missing parts in a design can be implemented in a way such that the complete design is equivalent to a given specification. For most instances, (approximate) QBF formulations returned inconclusive answers, and only DQBF was able to prove unrealizability of the incomplete design. Recently, Fröhlich et al. [16] presented the DQBF solver iDQ, which applies similar instantiation-based techniques as used in state-of-the-art decision procedures for effectively propositional logics [17]; it is the first publically available DQBF solver.

In this paper we present an elimination-based strategy for solving DQBF, accompanied by several optimizations. We developed an elimination strategy that eliminates a minimum set of variables that cause the non-linear dependencies. Hence, the strategy guides the elimination routine in order to obtain an easier-to-solve QBF instance. Once there are only linear dependencies left, we employ a QBF solver for the remaining instance. The elimination routine for both QBF and DQBF is based on And-Inverter-Graphs (AIGs) [18], [19]. For this data structure we introduce an efficient algorithm for pure and unit literal detection, which is employed between the elimination routines.

As reference application we use the analysis of incomplete digital circuits as in [15], [10], [16]. These circuits contain parts which are not yet implemented: so-called black boxes. An important question for an incomplete design is its realizability, which is also known as the partial equivalence checking problem (PEC) [20]. Incomplete designs with black boxes come into play, if already in an early stage of development, when not all modules are available yet, verification techniques are to be employed to find errors in the finished parts. Circuits can also be incomplete because parts have been removed that are notoriously hard to verify like multipliers or large memories, but that are expected not to influence the property to be checked. Also for diagnostic purposes, the removal of parts of a circuit can be beneficial. Previous SAT- and QBF-based approaches to the PEC problem fail to provide accurate answers in case the design contains more than one black box: Since the quantifiers in QBF are linearly ordered, the exact dependencies of the black boxes on different subsets of the circuit's signals cannot be expressed [10], such that QBF can only serve as an approximate decision method (like SAT as well).

In our experiments we show that our elimination-based approach improves existing ones significantly: Our solver HQS ("Henkin

978-3-9815370-4-8/DATE15/©2015 EDAA

Quantified Solver") is able to validate 50 % more benchmark instances than iDQ on the given benchmark set and speeds up the computation time by up to four orders of magnitude.

Organization of the paper: In the following section we briefly introduce the necessary foundations. In Section III we present our solution strategy for DQBF. An experimental evaluation and comparison with iDQ follows in Section IV. The final Section V concludes the paper and gives directions for future research.

II. Foundations

In this section, we briefly review the foundations of DQBF, QBF, and And-Inverter Graphs (AIGs).

A. Dependency quantified Boolean formulas

Definition 1 (DQBF syntax) *Let $V = \{x_1, \ldots, x_n, y_1, \ldots, y_m\}$ be a set of Boolean variables. A* dependency quantified Boolean formula (DQBF) *ψ over the variables V has the following form:*

$$\forall x_1 \forall x_2 \ldots \forall x_n \exists y_1(D_{y_1}) \exists y_2(D_{y_2}) \ldots \exists y_m(D_{y_m}) : \varphi,$$

where $D_{y_i} \subseteq \{x_1, \ldots, x_n\}$ for $i = 1, \ldots, m$ and φ is a Boolean formula over V. The set D_{y_i} is called the dependency set *of y_i, and $\exists y_i(D_{y_i})$ is also called a* Henkin quantifier *[9].*

The part $\forall x_1 \ldots \exists y_m(D_{y_m})$ is the *quantifier prefix* of ψ and φ its *matrix*. Often it is assumed w. l. o. g. that φ is given in conjunctive normal form (CNF), i. e., that it is a conjunction of clauses, which are disjunctions of variables or their negations (literals). We fix the sets $V_\psi^\exists = \{y_1, \ldots, y_m\}$ of existential and $V_\psi^\forall = \{x_1, \ldots, x_n\}$ of universal variables of ψ.

For a set $V' \subseteq V$ of variables, let $\mathcal{A}(V') = \{v : V' \to \{0,1\}\}$ denote the set of assignments of the variables in V'.

Definition 2 (DQBF semantics) *Let $\psi = \forall x_1 \ldots \forall x_n \exists y_1(D_{y_1}) \ldots \exists y_m(D_{y_m}) : \varphi$ be a DQBF. It is* satisfied *iff for all $i = 1, \ldots, m$ there are Boolean functions $s_{y_i} : \mathcal{A}(D_{y_i}) \to \{0,1\}$ such that replacing each y_i in φ by (a Boolean expression for) s_{y_i} yields a tautology. The function s_{y_i} is called a* Skolem function *for y_i.*

Currently there are three solving techniques known in the literature for validating DQBF: 1) search-based [14], 2) elimination-based [10], and 3) instantiation-based [16]. In this paper, we utilize an elimination-based algorithm using AIGs (cf. Section II-C) as in [10].

An important special case of a DQBF is a QBF, in which the dependencies of the existential variables on the universal ones induce a linear ordering. Its syntax is typically defined as follows:

Definition 3 (QBF syntax) *Let $V = \{x_1, \ldots, x_n, y_1, \ldots, y_m\}$ be a set of Boolean variables. A* quantified Boolean formula (QBF) *Ψ over the variables V has the following form:*

$$\Psi = \forall X_1 \exists Y_1 \forall X_2 \exists Y_2 \ldots \forall X_k \exists Y_k : \varphi,$$

where $X_1, \ldots, X_k$ is a partition of $\{x_1, \ldots, x_n\}$ and $Y_1, \ldots, Y_k$ is a partition of $\{y_1, \ldots, y_m\}$ with $X_i \neq \emptyset$ for $i = 2, \ldots, k$ and $Y_j \neq \emptyset$ for $j = 1, \ldots, k-1$. The matrix φ is a Boolean formula over V.

Let Ψ be a QBF as in Definition 3. It is equivalent to the DQBF $\psi = \forall x_1 \ldots \forall x_n \exists y_1(D_{y_1}) \ldots \exists y_m(D_{y_m}) : \varphi$, where $D_{y_i} = \bigcup_{k=1}^{\ell} X_k$ with Y_ℓ being the unique set with $y_i \in Y_\ell$.

Example 1 *Consider the following DQBF:*

$$\psi = \forall x_1 \forall x_2 \exists y_1(x_1) \exists y_2(x_2) : \varphi\,.$$

The existential variables y_1 and y_2 depend only on one universal variable. There is no QBF prefix exactly representing these dependencies.

Deciding whether a DQBF is satisfied is NEXPTIME-complete [8], whereas deciding a QBF is "only" PSPACE-complete [21].

B. Variable elimination for DQBF

Elimination-based methods for propositional formulas are based on the elimination of certain variables by combining cofactors of the formula in an appropriate way. The authors in [10] present a DQBF algorithm based on variable elimination of universal literals.

For a variable $v \in V$ and a Boolean expression ψ over $V \setminus \{v\}$, let $\varphi[\psi/v]$ denote the expression which results from replacing all occurrences of v in φ by ψ.

Theorem 1 (Elimination of universal variables, [10]) *Let $E_{x_i} = \left\{ y_j \in V_\psi^\exists \,\middle|\, x_i \in D_{y_j} \right\}$ be the set of existential variables which depend on the universal variable x_i. Then ψ is equivalent to the following DQBF:*

$$\begin{aligned}\psi' := \forall x_1 \ldots \forall x_{i-1} \forall x_{i+1} \ldots \forall x_n \\ \exists y_1(D_{y_1} \setminus \{x_i\}) \ldots \exists y_m(D_{y_m} \setminus \{x_i\}) \underbrace{\exists y'_j(D_{y_j} \setminus \{x_i\})}_{\text{for all } y_j \in E_{x_i}} : \\ \varphi[0/x_i] \wedge \varphi[1/x_i][y'_j/y_j \ \forall y_j \in E_{x_i}].\end{aligned}$$

This elimination rule allows us to replace the DQBF at hand by an equivalent one which contains one universal variable less—until a pure SAT formula is obtained.

In general, the universal elimination comes at the price of additional existential variables, which can be eliminated (like in QBF) if they depend on all universal variables occurring in the formula:

Theorem 2 (Elimination of existential variables, [10]) *Consider the following DQBF:*

$$\psi := \forall x_1 \ldots \forall x_n \exists y_1(D_{y_1}) \ldots \exists y_m(D_{y_m}) : \varphi$$

If $D_{y_i} = V_\psi^\forall$, i. e., if y_i depends on all universal variables, ψ is equivalent to:

$$\begin{aligned}\forall x_1 \ldots \forall x_n \exists y_1(D_{y_1}) \ldots \exists y_{i-1}(D_{y_{i-1}}) \\ \exists y_{i+1}(D_{y_{i+1}}) \ldots \exists y_m(D_{y_m}) : \varphi[0/y_i] \vee \varphi[1/y_i].\end{aligned}$$

The algorithm in [10] uses the elimination of existential variables whenever possible. Otherwise, all universal variables are eliminated until a SAT instance remains, which is validated by a standard SAT-solver.

C. And-inverter graphs

And-inverter graphs (AIGs) [22], [18] are Boolean circuits composed solely of AND gates with two inputs and inverters. They can be used for representing arbitrary Boolean formulas. In contrast to BDDs [23], AIGs are not canonical—for each propositional formula several structurally different AIG representations can exist—allowing them to be potentially more compact than BDDs.

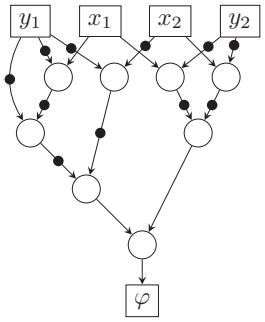

Fig. 1. Example of an AIG for the function $\varphi = (y_1 \vee x_1) \wedge (y_1 \vee x_2) \wedge (y_2 \vee \neg x_1) \wedge (y_2 \vee \neg x_2)$

Example 2 *In Fig. 1 a small AIG is shown. The white nodes represent the AND gates and the small black nodes the inverter gates. The AIG represents the function:*

$$\varphi = \left(\overline{\left(\overline{(\overline{y_1} \wedge x_1)} \wedge \overline{y_1} \right)} \wedge \overline{(\overline{y_1} \wedge \overline{x_2})} \right) \wedge \left(\overline{(x_1 \wedge \overline{y_2})} \wedge \overline{(x_2 \wedge \overline{y_2})} \right)$$

For readability reasons we write $\overline{\chi}$ instead of $\neg\chi$. By simple transformations one can see that φ is equivalent to the CNF $(y_1 \vee x_1) \wedge (y_1 \vee x_2) \wedge (y_2 \vee \neg x_1) \wedge (y_2 \vee \neg x_2)$.

A special case of AIGs are so-called functionally reduced and-inverter graphs (FRAIGs) [24]. In contrast to AIGs a FRAIG is "pseudo-canonical", i. e., there are no two distinct gates in the FRAIG computing the same (or inverse) function. However, FRAIGs still allow multiple structurally different representations of the same function. FRAIGs efficiently support conjunction, disjunction and composition of Boolean functions as well as existential and universal quantification of a single Boolean variable. Hence, FRAIGs have successfully been employed in decision procedures for QBF [19] and DQBF [10]. In the following we will always refer to AIGs in the meaning of "AIG or FRAIG", since many operations are performed on AIGs followed by a conversion to FRAIGs from time to time.

III. Improving DQBF solving

In this section we describe our improved method for elimination-based DQBF solving, based on two cornerstones: 1) a sophisticated heuristics to determine the next variable to be eliminated based on dependency graphs (cf. Section III-A), and 2) pure literal detection on AIGs (cf. Section III-B). Furthermore, we utilize techniques known from QBF to improve the scalability. The complete algorithm is summarized in Section III-C.

A. From DQBF to QBF

For QBFs, which exhibit a linearly ordered quantifier prefix, many efficient solvers exist, both search-based (e. g., DepQBF [25]) as well as based on variable elimination (e. g., AIGSOLVE [26], [19]). In order to benefit from the progress these solvers made, we strive for turning the DQBF at hand into an equivalent QBF by eliminating a small (or even minimum) set of universal variables which are responsible for the DQBF's non-linear dependencies.

Definition 4 *Let $\psi = \forall x_1 \dots \forall x_n \exists y_1(D_{y_1}) \dots \exists y_m(D_{y_m}) : \varphi$ be a DQBF. The* dependency graph *of ψ is the directed graph $G_\psi = (V, E)$ with $V = \{y_1, \dots, y_m\}$ and $E = \left\{(y_i, y_\ell) \in V \times V \,\middle|\, D_{y_i} \not\subseteq D_{y_\ell}\right\}$.*

Fig. 2. Example of a dependency graph for $\forall x_1 \forall x_2 \exists y_1(x_1) \exists y_2(x_2) : \varphi$

The intuition is that there is an edge from y_i to y_ℓ iff y_i depends on some universal variables on which y_ℓ does not. This implies that if there is an equivalent QBF prefix, then y_i has to be placed right of y_ℓ with the variables from $D_{y_i} \setminus D_{y_\ell}$ in between.

Based on dependency graphs and this observation we can detect whether there is an equivalent QBF prefix for a given DQBF.

Theorem 3 *A DQBF ψ has an equivalent QBF prefix iff G_ψ is acyclic.*

The correctness of this theorem can be seen as follows: On the one hand, if there is an equivalent QBF prefix, we can order the existential variables according to their position in this QBF prefix. The edges of the dependency graph then all point from right to left according to this order. Therefore there cannot be a loop in the dependency graph.

On the other hand, if the dependency graph is acyclic, we can partition the existential variables: the first block Y_1 contains all variables without out-going edges. Then these variables are removed from the graph. This induces a new set of variables without out-going edges, which are placed in the second block Y_2. This operation is continued with blocks $Y_3, \dots, Y_k$ until the graph becomes empty. An equivalent QBF prefix is then $\forall X_1 \exists Y_1 \forall X_2 \dots \exists Y_k \forall X_{k+1}$ where X_i equals $D_y \setminus (X_1 \cup \dots \cup X_{i-1})$ if $y \in Y_i$. Note that all variables in the same block Y_i have the same dependency set. Additionally $X_{k+1} = V_\psi^\forall \setminus (X_1 \cup \dots \cup X_k)$.

A formal proof is contained in the extended version [27] of this paper.

Example 3 *Consider the DQBF from Example 1: $\psi = \forall x_1 \forall x_2 \exists y_1(x_1) \exists y_2(x_2) : \varphi$. As discussed in the example this DQBF has no equivalent QBF prefix. The corresponding dependency graph is given in Fig. 2. From Theorem 3 we see that there is indeed no equivalent QBF prefix due to the cycle.*

The following lemma and theorem lead us to a mechanism for obtaining a QBF, given an arbitrary DQBF.

Lemma 1 *Let ψ be a DQBF as before and $G_\psi = (V, E)$ its dependency graph. Let $y_1 \to y_2 \to \dots \to y_k \to y_{k+1} = y_1$ be a cycle in G_ψ of length k. Then there is $i \in \{1, \dots, k\}$ such that $y_{i+1} \to y_i$.*

Proof: Let the cycle be given as above. Assume $(y_{i+1}, y_i) \notin E$ for all $1 \le i \le k$. That means, $D_{y_{i+1}} \subseteq D_{y_i}$. Due to the transitivity of the subset relation we have

$$D_{y_1} = D_{y_{k+1}} \subseteq D_{y_k} \subseteq D_{y_{k-1}} \subseteq \dots \subseteq D_{y_1} .$$

This implies $D_{y_1} = D_{y_2} = \dots = D_{y_k}$.

On the other hand, $y_i \to y_{i+1}$ implies $D_{y_i} \not\subseteq D_{y_{i+1}}$ for all $i = 1, \dots, k$, i. e., $D_{y_i} \neq D_{y_{i+1}}$, which is a contradiction. ■

Theorem 4 *For a DQBF ψ, its dependency graph G_ψ is cyclic iff there are y_i, y_k such that $D_{y_i} \not\subseteq D_{y_k}$ and $D_{y_k} \not\subseteq D_{y_i}$.*

Proof: Lemma 1 implies that if G_ψ is cyclic, then there are y, y' with $y \to y'$ and $y' \to y$. The other direction is trivial. ■

To check whether a DQBF is actually expressible as QBF, one only has to consider the dependencies between pairs of variables. This property can be used to determine a minimum set of universal variables that have to be eliminated to obtain an equivalent QBF, which can be solved by an arbitrary QBF solver.

We express the problem to determine such a minimum set as a partial MaxSAT problem. MaxSAT is an extension to SAT, determining the maximum number of simultaneously satisfied clauses of a Boolean formula in CNF. Partial MaxSAT is a variation where clauses can be defined as hard or soft. Hard clauses *must* be satisfied, otherwise the MaxSAT instance is unsatisfiable (as in pure SAT), and soft clauses *may* be satisfied (as in pure MaxSAT). Many graph theoretic problems, such as the Max-flow-min-cut problem can be expressed by MaxSAT or its variations. The interested reader is referred to [1] for more details about MaxSAT.

Our partial MaxSAT instance to determine the minimum dependency set to be eliminated is built as follows: For each universal variable $x \in V_\psi^\forall$ we introduce a variable $\hat{x}$ of the MaxSAT problem such that for an optimal solution $\hat{x} = 1$ means that x belongs to the set of variables to be eliminated.

For each binary loop consisting of existential variables $y, y' \in V_\psi^\exists$ with dependency sets $D_y, D_{y'}$ we have to eliminate universal variables such that D_y becomes a subset of $D_{y'}$ or vice versa. That means we either have to eliminate all variables in $D_y \setminus D_{y'}$ or all in $D_{y'} \setminus D_y$. This can be expressed as partial MaxSAT as follows:

Let $C_\psi := \big\{\{y, y'\} \subseteq V_\psi^\exists \,\big|\, D_y \not\subseteq D_{y'} \wedge D_{y'} \not\subseteq D_y\big\}$ be the set of binary cycles representing the hard constraint:

$$\zeta_\psi^{\text{hard}} := \bigwedge_{\{y,y'\} \in C_\psi} \Big(\bigwedge_{x \in D_y \setminus D_{y'}} \hat{x} \;\vee \bigwedge_{x \in D_{y'} \setminus D_y} \hat{x} \Big). \tag{1}$$

The soft constraint for optimization is given by

$$\zeta_\psi^{\text{soft}} := \bigwedge_{x \in V_\psi^\forall} \neg\hat{x}\,. \tag{2}$$

The MaxSAT solver determines an assignment ν of the variables such that ζ_ψ^{hard} is satisfied and a maximum number of variables is assigned to 0. The variables to be eliminated are given by $\{x \in V_\psi^\forall \,|\, \nu(\hat{x}) = 1\}$.

B. Unit and Pure Variables

The detection of unit and pure literals is a fundamental technique for search-based QBF solvers which work on a formula in CNF.

The following definition holds for any propositional formula and defines a semantic criterion for a unit or pure variable.

Definition 5 (Unit and pure variables) *Let φ be a propositional formula over the variable set V. A variable $v \in V$ is* positive (negative) unit *in ψ, if $\varphi[0/v]$ ($\varphi[1/v]$, resp.) is unsatisfiable.*

The variable $v \in V$ is positive (negative) pure *in ψ, if $\varphi[0/v] \wedge \neg\varphi[1/v]$ ($\varphi[1/v] \wedge \neg\varphi[0/v]$, resp.) is unsatisfiable.*

In case we have detected a unit or pure variable, it can be eliminated as given in the following theorem (cf. also [16]):

Theorem 5 (Elimination of unit and pure variables) *Let $\psi = Q : \varphi$ be a DQBF over V and $v \in V$. We denote the quantifier prefix which results from Q by removing all occurrences of v by $Q \setminus \{v\}$.*

- *If v is existentially quantified and positive (negative) unit, then ψ is equivalent to $Q \setminus \{v\} : \varphi_{|v=1}$ $_{(=0)}$.*
- *If v is universally quantified and positive or negative unit, then ψ is unsatisfied.*
- *If v is existentially quantified and positive (negative) pure, then ψ is equivalent to $Q \setminus \{v\} : \varphi_{v=1}$ $_{(=0)}$.*
- *If v is universally quantified and positive (negative) pure, then ψ is equivalent to $Q \setminus \{v\} : \varphi_{v=0}$ $_{(=1)}$.*

Proof: A proof can be found in the extended version [27] of this paper. ■

QBF solvers employ a sufficient (but not neccessary) syntactic criterion to determine unit and pure variables efficiently. A sufficient *and* necessary check is usually too expensive, as in general it requires a check for satisfiability for each variable.

Lemma 2 (Unit and pure variables in CNF) *Given a Boolean formula φ over variables V in CNF and $v \in V$, the variable v is unit, if there exists a clause in φ consisting only of v or of $\neg v$. The variable v is pure, if v occurs either only positive or only negative in the whole CNF.*

In search-based solvers this check is performed dynamically by considering the CNF resulting from temporarily assigned variables.

The elimination of unit and pure variables from a formula is particularly beneficial for DQBF because it does not require the duplication of any variables, but rather reduces both the number of variables and the size of the AIG.

In contrast to [26] where semantic checks for unit variables are used as a preprocessing step, we profit from unit and pure variables in an inner loop of our algorithm and therefore we prefer a syntactic check in order to avoid SAT solver calls. Unfortunately, the clause structure is destroyed when using AIGs for representing the matrix, and thus these rather cheap syntactic checks using clauses cannot be applied within our framework. Instead we exploit the following syntactic criterion for AIGs:

Theorem 6 (Unit and pure variables in AIGs) *Let ψ be a DQBF over variables V with matrix φ and assume that φ is represented by an AIG. Let n_v be the input node corresponding to $v \in V$ and n_φ the output node corresponding to φ.*

If there is a path from n_v to n_φ without negation then v is positive unit. If there is a path from n_v to n_φ such that the only negation on this path appears between n_v and the first AND node, then v is negative unit.

If the number of negations on all paths from n_v to n_φ is even, then v is positive pure. If the number of negations is odd on all paths, v is negative pure.

Proof: This theorem is proven by induction on the structure of the AIG in the extended version [27] of this paper. ■

By a recursive traversal of the AIG we can determine in $O(|\varphi| + |V|)$ variables which are unit or pure according to Theorem 6. Here $|\varphi|$ denotes the number of AND-nodes in the AIG-representation of φ and $|V|$ the number of variables it depends on. Unit and pure variables are eliminated using Theorem 5.

Example 4 *Consider again the AIG in Fig. 1. The syntactic check for purity identifies variable y_2 as positive pure because both paths from y_2 to φ contain an even number of inverters (2 inverters each). The syntactic check fails for the other three variables.*

However, as the AIG represents the function $(y_1 \vee x_1) \wedge (y_1 \vee x_2) \wedge (y_2 \vee \neg x_1) \wedge (y_2 \vee \neg x_2)$, it is easy to see that y_1 is positive pure according to Lemma 2, which is not detected by the syntactic AIG check.

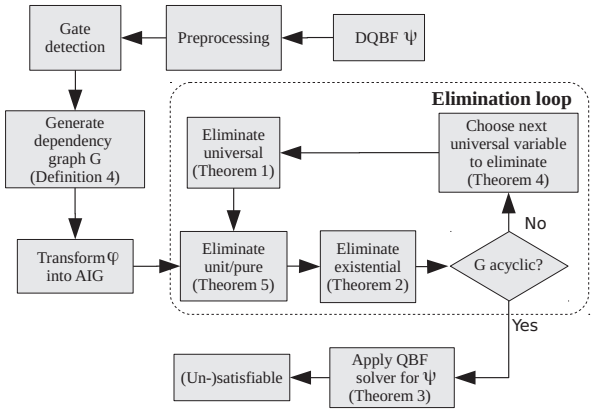

Fig. 3. Algorithmic Flow

Like on CNFs the syntactic check on AIGs is incomplete, but—as experiments show—very fast and still detects the majority of all unit and pure variables.

C. Elimination-based algorithm

The elimination procedure including all improvements presented in this section is illustrated in Fig. 3.

First, we utilize some basic preprocessing steps on the CNF, which are known from QBF preprocessing, but have been adapted to the DQBF setting: Detection of Tseitin-encoded [28] gates [19], which is effective if the CNF was generated from a circuit, universal reduction [29], removal of equivalent variables, and unit literal propagation.

After propagating all unit literals, we use the generalized universal reduction rule for clauses [1, Section 23.5]. In QBF, a universal literal u can be deleted from a clause if no existential literal in the clause depends on u, which can be naturally extended to DQBF [13, Section 4], [16]. Additionally we detect equivalent variables by analyzing the binary clauses of the formula and perform corresponding replacements. We apply these techniques in alternation until the CNF does not change anymore.

As the last preprocessing step we apply gate detection: Tseitin-encoding introduces an auxiliary variable for each gate output and adds clauses which encode the relationship between gate inputs and the gate output. We detect these clauses for AND, OR, and XOR gates (with arbitrarily negated inputs), remove the clauses for encoding the relationship between gate inputs and output, and store the relationship directly. The result is a CNF plus a list of gates. After preprocessing we create an AIG representation from the CNF. In this AIG, we replace all literals representing a gate output by the function computed by its gate using the `compose` operation on AIGs. In particular, there is no need for explicit elimination of the internal auxiliary variables resulting from the Tsetin encoding. The interested reader is referred to [19] for further details.

Before the main solving loop starts, we determine a dependency graph G_ψ (Definition 4), as described in Section III-A. Based on G_ψ we use the MaxSAT formulation of Equations 1 and 2 to compute a minimal set of universal variables whose elimination leads to a QBF problem. These universal variables are ordered according to the number of copies of existential variables which would be introduced by an elimination according to Theorem 1. However, we do not perform these eliminations immediately. Rather, in the main solving loop we first check for pure and unit literals of the current AIG (cf. Theorems 5 and 6). Additionally we eliminate all existential variables depending on all universal variables using Theorem 2.

In case G_ψ is still cyclic, we then choose the next universal variable from the ordered list computed above and eliminate it using Theorem 1.

Finally, when G_ψ becomes acyclic, we build a QBF upon the linearized dependency structure of the DQBF and employ a QBF solver. Here, we utilize AIGSOLVE [26], which also uses AIGs, thus we can feed the remaining AIG directly into this solver instead of transforming the AIG back into CNF.

Note, we also can stop the main loop if at any time an AIG representation of the constant 0 (or 1) function is obtained (see also [10]). Variables which do not occur in the support of the matrix can be removed from the quantifier prefix.

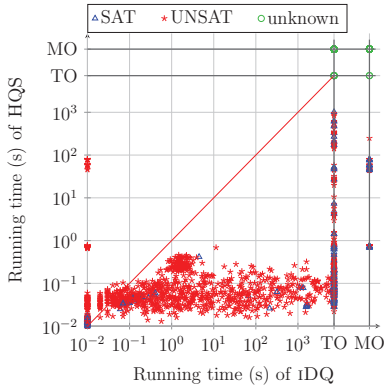

Fig. 4. Comparison of iDQ and HQS on all 1820 instances

IV. Experimental evaluation

We created a prototypic implementation in C++ of the previously described techniques, called HQS. We used the library `aigpp` [18] for the manipulation of AIGs and AIGSOLVE [26] as a QBF solver. `antom` [30] serves as the solver for partial MaxSAT. For comparison, we took the only available DQBF solver iDQ by Fröhlich et al. [16].

For evaluation we used the same 1100 PEC instances as [15] and [16]. These instances are PEC problems encompassing *adders*, arbiter implementations *lookahead* and *bitcell* described in [31] as well as the circuit family *pec_xor* from [15].[1] The additional $3 \cdot 240$ benchmarks *Z4*, *comp*, and *C432* consist also of PEC problems on circuits from the ISCAS 85 circuit library (cf. [32]).

All experiments were run on one Intel Xeon E5-2650v2 core at 2.60 GHz, with 64 GB of main memory and with Ubuntu Linux 12.04 in 64-bit mode as operating system. We aborted all experiments whose computation time exceeded two hours or which required more than 8 GB of memory.

Figure 4 compares the running times of iDQ and HQS on the considered 1820 benchmark instances. Note that the axes are in logarithmic scale. A marker below the diagonal means that HQS was faster than iDQ, a marker above it that iDQ was faster. A marker on the vertical (horizontal) lines denoted "TO" and "MO" indicate that iDQ (HQS) ran out of time (memory, resp.). The runtimes of HQS are clearly superior for almost all instances, on some instances by four orders of magnitude. HQS solves all instances solved by iDQ and 520 additional ones. An overview on the results for the different benchmark classes is given in Table I. There we give the number of instances in a

[1] In our presentation of the results we omit the 100 *pec_xor*-2 instances as both iDQ and HQS solved each instance in less than 0.05 seconds.

TABLE I
EXPERIMENTAL RESULTS

		HQS					iDQ				
Benchmark	#instances	solved	(SAT/UNSAT)	unsolved	(TO/MO)	total time	solved	(SAT/UNSAT)	unsolved	(TO/MO)	total time
adder	300	300	(42/258)	0	(0/0)	9.72	216	(3/213)	84	(84/0)	89,827.94
bitcell	300	300	(7/293)	0	(0/0)	11.27	190	(2/188)	110	(110/0)	78,106.86
lookahead	300	300	(10/290)	0	(0/0)	23.17	273	(4/269)	27	(27/0)	39,540.15
pec_xor	200	200	(24/176)	0	(0/0)	33.60	200	(24/176)	0	(0/0)	181.58
z4	240	240	(72/168)	0	(0/0)	4.86	111	(8/103)	129	(129/0)	41,626.30
comp	240	155	(39/116)	85	(9/76)	17.82	25	(0/25)	215	(180/35)	11.60
C432	240	60	(19/41)	180	(0/180)	1,332.58	20	(0/20)	220	(85/135)	0.20
total	1,820	1,555	(213/1,342)	265	(9/256)	1,433.02	1,035	(41/994)	785	(615/170)	249,294.63

class and for both solvers the numbers of solved and unsolved instances, additionally separated into SAT/UNSAT instances and timout/memout. The columns "total time" give the accumulated running time (in seconds) of the respective solver on those instances which were solved by both solvers. It should be noted that HQS solves 1413 of 1555 solved instances ($\approx 90\,\%$) in less than 1 second (iDQ only 507 of 1035 instances). The time for solving the MaxSAT problem for choosing the variables to eliminate was below 0.06 seconds for all instances. The time for syntactic unit/pure checks was less than 4 % of the runtime of each instance.

There are a few instances on which iDQ is faster, in particular among the *z4*, *comp*, and *C432* instances. For these instances, iDQ needs only a single SAT solver call to detect unsatisfiability—and therefore very little time. We could integrate such a SAT solver call into our preprocessing routine; this would reduce the running times for some instances without increasing it measurably for other instances. A more detailed evaluation can be found in the extended version of this paper [27].

V. CONCLUSION

This paper has presented a quantifier-elimination strategy based on dependency graphs which can be used to solve DQBF. Together with optimizations like simple preprocessing steps, the efficient detection of unit and pure variables on AIGs, and turning the DQBF into a QBF by eliminating a minimum set of universal variables, this constitutes a decision procedure which is clearly superior to the available methods both in computation time and the number of solved instances.

Future work will concentrate on more sophisticated preprocessing techniques and improvements on the choice and order of variables to be eliminated.

REFERENCES

[1] A. Biere, M. Heule, H. van Maaren, and T. Walsh, Eds., *Handbook of Satisfiability*, ser. Frontiers in Artificial Intelligence and Applications. IOS Press, 2009, vol. 185.

[2] E. M. Clarke, A. Biere, R. Raimi, and Y. Zhu, "Bounded model checking using satisfiability solving," *Formal Methods in System Design*, vol. 19, no. 1, pp. 7–34, 2001.

[3] F. Ivancic, Z. Yang, M. K. Ganai, A. Gupta, and P. Ashar, "Efficient SAT-based bounded model checking for software verification," *Theoretical Computer Science*, vol. 404, no. 3, pp. 256–274, 2008.

[4] A. Czutro, I. Polian, M. D. T. Lewis, P. Engelke, S. M. Reddy, and B. Becker, "Thread-parallel integrated test pattern generator utilizing satisfiability analysis," *Int'l Journal of Parallel Programming*, vol. 38, no. 3-4, pp. 185–202, 2010.

[5] S. Eggersglüß and R. Drechsler, "A highly fault-efficient SAT-based ATPG flow," *IEEE Design & Test of Computers*, vol. 29, no. 4, pp. 63–70, 2012.

[6] J. Rintanen, K. Heljanko, and I. Niemelä, "Planning as satisfiability: parallel plans and algorithms for plan search," *Artificial Intelligence*, vol. 170, no. 12-13, pp. 1031–1080, 2006.

[7] C. Jordan and M. Seidl, "QBF gallery," 2014, http://qbf.satisfiability.org/gallery/index.html.

[8] G. Peterson, J. Reif, and S. Azhar, "Lower bounds for multiplayer non-cooperative games of incomplete information," *Computers & Mathematics with Applications*, vol. 41, no. 7–8, pp. 957–992, 2001.

[9] L. Henkin, "Some remarks on infinitely long formulas," in *Infinitistic Methods: Proc. of the 1959 Symp. on Foundations of Mathematics*. Pergamon Press, 1961, pp. 167–183.

[10] 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. of ICCD*. IEEE CS, 2013, pp. 396–403.

[11] C. M. Wintersteiger, Y. Hamadi, and L. M. de Moura, "Efficiently solving quantified bit-vector formulas," in *Proc. of FMCAD*. IEEE, 2010, pp. 239–246.

[12] G. Kovásznai, A. Fröhlich, and A. Biere, "On the complexity of fixed-size bit-vector logics with binary encoded bit-width," in *Proc. of SMT@IJCAR'12*, ser. EPiC Series, vol. 20. EasyChair, 2013, pp. 44–56.

[13] V. Balabanov, H.-J. K. Chiang, and J.-H. R. Jiang, "Henkin quantifiers and Boolean formulae – a certification perspective of DQBF," *Theoretical Computer Science*, vol. 523, 2014, 86–100.

[14] A. Fröhlich, G. Kovásznai, and A. Biere, "A DPLL algorithm for solving DQBF," in *Proc. of the Int'l Workshop on Pragmatics of SAT (POS)*, 2012.

[15] B. Finkbeiner and L. Tentrup, "Fast DQBF refutation," in *Proc. of SAT*, ser. LNCS, vol. 8561. Springer, Jul. 2014, pp. 243–251.

[16] A. Fröhlich, G. Kovásznai, A. Biere, and H. Veith, "iDQ: Instantiation-based DQBF solving," in *Proc. of the Int'l Workshop on Pragmatics of SAT (POS)*, Vienna, Austria, 2014.

[17] K. Korovin, "Inst-Gen – a modular approach to instantiation-based automated reasoning," in *Programming Logics – Essays in Memory of Harald Ganzinger*, ser. LNCS, vol. 7797. Springer, 2013, pp. 239–270.

[18] F. Pigorsch, C. Scholl, and S. Disch, "Advanced unbounded model checking based on AIGs, BDD sweeping, and quantifier scheduling," in *Proc. of FMCAD*. IEEE Computer Society Press, Nov. 2006, pp. 89–96.

[19] F. Pigorsch and C. Scholl, "Exploiting structure in an AIG-based QBF solver," in *Proc. of DATE*. IEEE, 2009, pp. 1596–1601.

[20] C. Scholl and B. Becker, "Checking equivalence for circuits containing incompletely specified boxes," in *Proc. of ICCD*. IEEE CS, 2002, pp. 56–63.

[21] L. J. Stockmeyer, "The polynomial-time hierarchy," *Theoretical Computer Science*, vol. 3, no. 1, pp. 1–22, 1976.

[22] A. Kuehlmann, M. K. Ganai, and V. Paruthi, "Circuit-based Boolean Reasoning," in *Proc. of DAC*, 2001.

[23] R. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," *IEEE Trans. on Comp.*, vol. 35, no. 8, pp. 677–691, 1986.

[24] A. Mishchenko, S. Chatterjee, R. Jiang, and R. K. Brayton, "FRAIGs: A unifying representation for logic synthesis and verification," EECS Dept., UC Berkeley, Tech. Rep., 03 2005.

[25] F. Lonsing and A. Biere, "DepQBF: A dependency-aware QBF solver," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 7, no. 2-3, pp. 71–76, 2010.

[26] F. Pigorsch and C. Scholl, "An AIG-based QBF-solver using SAT for preprocessing," in *Proc. of DAC*. ACM Press, 2010, pp. 170–175.

[27] K. Gitina, R. Wimmer, S. Reimer, B. Becker, and C. Scholl, "Solving DQBF through quantifier elimination (extended version)," available at http://www.avacs.org, Reports of the SFB/TR14 AVACS, Tech. Rep. ATR 107, Dec. 2014.

[28] G. S. Tseitin, "On the complexity of derivation in propositional calculus," *Studies in Constructive Mathematics and Mathematical Logic*, vol. Part 2, pp. 115–125, 1970.

[29] E. Giunchiglia, P. Marin, and M. Narizzano, "sQueezeBF: An effective preprocessor for QBFs based on equivalence reasoning," in *Proc. of SAT*, ser. LNCS. Springer, 2010, vol. 6175, pp. 85–98.

[30] T. Schubert and S. Reimer, "antom," 2014, https://projects.informatik.uni-freiburg.de/projects/antom.

[31] W. J. Dally and R. C. Harting, *Digital Design: A Systems Approach*. Cambridge University Press, 2012.

[32] C. Scholl and B. Becker, "Checking equivalence for partial implementations," in *Proc. of the 38th Design Automation Conference (DAC)*. ACM Press, 2001, pp. 238–243.