HQSpre – An Effective Preprocessor for QBF and DQBF*

Ralf Wimmer, Sven Reimer, Paolo Marin, and Bernd Becker

Albert-Ludwigs-Universität Freiburg, Freiburg im Breisgau, Germany
{wimmer,reimer,marin,becker}@informatik.uni-freiburg.de

Abstract. We present our new preprocessor HQSpre, a state-of-the-art tool for simplifying quantified Boolean formulas (QBFs) and the first available preprocessor for dependency quantified Boolean formulas (DQBFs). The latter are a generalization of QBFs, resulting from adding so-called Henkin-quantifiers to QBFs. HQSpre applies most of the preprocessing techniques that have been proposed in the literature. It can be used both as a standalone tool and as a library. It is possible to tailor it towards different solver back-ends, e. g., to preserve the circuit structure of the formula when a non-CNF solver back-end is used. Extensive experiments show that HQSpre allows QBF solvers to solve more benchmark instances and is able to decide more instances on its own than state-of-the-art tools. The same impact can be observed in the DQBF domain as well.

1 Introduction

Solvers for Boolean formulas have proven to be powerful tools for many applications, ranging from CAD, e. g., for formal verification [4] and circuit test [11], to artificial intelligence [33]. They are not only of academic interest, but have also gained acceptance in industry. While solvers for deciding satisfiability of quantifier-free propositional formulas (the famous SAT-problem [5]) have reached a certain level of maturity during the last years, solving quantified and dependency quantified Boolean formulas (QBFs and DQBFs [27], resp.) is still a hot topic in research. In particular, the last two decades have brought enormous progress in solving QBFs [29,16,25,19] and the last five years also in solving DQBFs [13,12,15]. With increasing improvements of solver technology also new applications have arisen which could not be handled (or only handled approximately) before, such as verification of partial designs [35,14], controller synthesis [7], and games with incomplete information [27].

One part of this success is due to improved solution methods not only based on depth-first search (the QDPLL algorithm) as implemented in solvers like DepQBF [25], QuBE [16], and AQuA (see [32]), but also using quantifier elimination as applied by AIGSolve [29] and Quantor [2], counterexample-guided abstraction refinement, which is the principle underlying the solvers

* This work was supported by the German Research Council (DFG) as part of the project "Solving Dependency Quantified Boolean Formulas" and by the Sino-German Center for Research Promotion (CDZ) as part of the project CAP (GZ 1023).

GhostQ [22], RAReQS [19], and Qesto [20], and further algorithms. Another protagonist are sophisticated preprocessing techniques. Their goal is to simplify the formula using algorithms of lower complexity (mostly polynomial) than the actual decision problem before the solver is called. This can reduce the overall computation time by orders of magnitude and, most interestingly, it can even make solving instances feasible which cannot be solved without preprocessing.

Many techniques have been proposed for preprocessing and implemented in different tools. One can distinguish between four main types of preprocessor routines: clause elimination, clause strengthening, variable elimination, and other formula modifications. We will discuss these categories and the corresponding techniques in Sect. 2.2. All of them yield an equisatisfiable formula, which is typically easier to solve than the original one.

Contribution. In this paper, we present the new tool HQSpre, which supports most preprocessing techniques for QBFs and extends them to DQBFs. It is the first available tool for preprocessing DQBFs. The available QBF tools like sQueezeBF and Bloqqer only have a subset of these techniques available. HQSpre can be used both as a standalone tool and as a library. It is designed to be easily extensible and adaptable to different solver back-ends. For instance, if the back-end solver is not CNF-based, but rather works on a circuit representation of the formula, the preprocessor takes care not to destroy this structure, e. g., by forbidding the application of clause elimination routines to clauses that encode circuit gates.

We provide an extensive experimental evaluation where we show that HQSpre is state of the art: *(1)* it enables state-of-the-art QBF solvers (AIGSolve [29], AQuA [32], Caqe [38], DepQBF [25], Qesto [20], and RAReQS [19]) to solve more instances in less time than using the alternative preprocessors sQueezeBF [17] and Bloqqer [6], and it is robust over different kinds of solvers; *(2)* HQSpre is very effective on DQBFs as well, as it is able to solve directly or to simplify into QBFs many formulas, and lets DQBF solvers decide several more problems.

HQSpre is available as an open source tool. The most recent version can be downloaded from:
`https://projects.informatik.uni-freiburg.de/projects/dqbf/files` .

Structure of this paper. In the following section, we introduce the necessary foundations and describe the different preprocessing techniques and how they are implemented in HQSpre. Sect. 3 contains the results of our experiments. Finally, in Sect. 4, we conclude this paper with an outlook on future work.

2 Preprocessing Techniques in HQSpre

2.1 Foundations

Let V be a set of Boolean variables. We consider (dependency) quantified Boolean formulas in *prenex conjunctive normal form (PCNF)*: a quantifier-free Boolean

formula is in CNF if it is a conjunction of clauses. A *clause* is a disjunction of literals, and a *literal* is either a variable $v \in V$ or its negation $\neg v$. We write clauses in the form $\{v_1, \ldots, v_n\}$ with literals v_i. A clause is called *unit* if it contains only one literal, and called *binary* if it contains two literals. We denote the size of a formula as the number of all literals in all clauses. A formula is in PCNF if it can be split into a quantifier prefix and a Boolean formula in CNF, the matrix of the formula.

Definition 1. *Let $V = \{v_1, \ldots, v_n\}$ be a set of Boolean variables and φ a quantifier-free Boolean formula over V. A* quantified Boolean formula (QBF) *ψ has the form $\psi = Q_1 v_1 \ldots Q_n v_n : \varphi$ with $Q_i \in \{\forall, \exists\}$ and $v_i \in V$ for $i = 1, \ldots, n$. $Q_1 v_1 \ldots Q_n v_n$ is called the* quantifier prefix *and φ the* matrix *of ψ.*

We denote universal variables with x, and existential ones with y. If the quantifier does not matter, we use v. Accordingly, ℓ is an arbitrary literal, $\ell^\exists$ a literal with an existential variable, and $\ell^\forall$ a universal literal. For a literal ℓ, we define $\mathrm{var}(\ell)$ as the corresponding variable, i. e., $\mathrm{var}(v) = \mathrm{var}(\neg v) = v$.

The quantifier prefix imposes a linear order on the variables. One can think of a QBF as a two-player game: one player assigns the existential variables, the other player the universal ones. The game proceeds turn-based according to the prefix from left to right: When it is the existential player's turn, he assigns the corresponding existential variable, and similarly for the universal player. The goal of the existential player is to satisfy the formula, the universal player wants to falsify it. The formula is satisfiable if the existential player has a winning strategy, i. e., if he can satisfy the formula no matter how the universal player assigns his variables.

Dependency quantified Boolean formulas are a generalization of QBFs. They are obtained syntactically by relaxing the requirement of a linearly ordered prefix and making the dependencies explicit, and semantically by restricting the knowledge of the players.

Definition 2. *Let $V = \{x_1, \ldots, x_n, y_1, \ldots, y_m\}$ be a set of Boolean variables and φ a quantifier-free Boolean formula over V in CNF. A* dependency quantified Boolean formula (DQBF) *Ψ has the form $\forall x_1 \ldots \forall x_n \exists y_1(D_{y_1}) \ldots \exists y_m(D_{y_m}) : \varphi$, where $D_{y_i} \subseteq \{x_1, \ldots, x_n\}$ for $i = 1, \ldots, m$ is the dependency set of y_i.*

In contrast to QBF, a DQBF can be considered as a game with partial information: The universal player assigns all universal variables in the beginning. The existential player assigns a value to each existential variable y based only on the assignment of the universal variables in the corresponding dependency set D_y.

A DQBF is equivalent to a QBF iff for all existential variables y, y' the condition $D_y \subseteq D_{y'}$ or $D_{y'} \subseteq D_y$ holds.

DQBFs are strictly more expressive than QBFs. While deciding satisfiability of QBFs is PSPACE complete [26], deciding DQBFs is NEXPTIME complete [27].

2.2 Preprocessing Techniques

The goal of preprocessing the formula before the actual solution process is to simplify the formula. Experience suggests that benefits of preprocessing increase with

Algorithm 1 Outline of the main preprocessing routine. Note that after each routine, a fast formula simplification procedure is called. Universal reduction and subsumption checks are performed for each added or modified clause.

```
Preprocess((D)QBF ψ = Q : φ)
begin
    simplify(ψ)                                                          (1)
    repeat                                                               (2)
        if iteration ≤ gateConvLoops then                                (3)
            gates ← gateDetection(ψ)                                     (4)
            gateSubstitutionAndRewriting(ψ, gates); simplify(ψ)          (5)
        end if                                                           (6)
        eliminateClauses(ψ); simplify(ψ)    ◁ Hidden/covered TE/SE/BCE   (7)
        selfsubsumingResolution(ψ); simplify(ψ)                          (8)
        variableEliminationByResolution(ψ); simplify(ψ)                  (9)
        syntacticConstants(ψ); simplify(ψ)                               (10)
        if first iteration then                                          (11)
            semanticConstants(ψ); simplify(ψ)                            (12)
            trivialMatrixChecks(ψ)                                       (13)
        end if                                                           (14)
        universalExpansion(ψ); simplify(ψ)                               (15)
    until ψ has not been changed anymore or ψ is decided                 (16)
    return ψ                                                             (17)
end
```

the difficulty of the decision problem. As mentioned already in the introduction, the techniques that we apply in our preprocessor HQSpre can be grouped into four different classes: (1) variable elimination, (2) clause elimination, (3) clause strengthening, and (4) other formula modification routines. Due to space restrictions, we cannot provide all details of the techniques. For more information we refer the reader to the cited literature. We present more details for routines which a) are not described or applied in the literature so far or b) have interesting implementation details.

Algorithm 1 gives an overview of the main preprocessing routine which calls the different techniques in a loop until the formula does not change anymore.

Variable Elimination Routines. We define variable elimination routines as methods which are able to remove a variable v from the formula φ. The first kind of such techniques can eliminate v whenever we can fix the truth value of v and propagate it through φ using *Boolean constraint propagation (BCP)* [5]. Common techniques are the detection of *constant* and *pure* literals [5]. For efficiency reasons both kinds are usually only checked syntactically: a literal ℓ is constant if there exists a unit clause $\{\ell\}$ in φ, and a literal ℓ is pure if φ does not contain $\neg\ell$ in any clause. Both conditions can be easily generalized to (D)QBF [40].

We also apply a check for constants using another syntactic criterion over binary clauses. For this, we first determine the transitive implication closure, i. e.,

binary clauses of the form: $\{\neg\ell_1, \ell_2\}$, $\{\neg\ell_2, \ell_3\}$, $\{\neg\ell_3, \ell_4\}$, $\ldots$, $\{\neg\ell_{n-1}, \ell_n\}$, and $\{\neg\ell_n, \neg\ell_1\}$. These clauses represent a chain of implications: if ℓ_1 is assigned the truth value $\top$, we can deduce that also $\ell_2, \ell_3, \ell_4, \ldots, \ell_n$ get the truth value $\top$ in the example above, and this in turn implies that ℓ_1 has to get the truth value $\bot$, i. e., ℓ_1 implies $\neg\ell_1$, which is a contradiction. So the matrix will be unsatisfied, if we set ℓ_1 to $\top$. Hence, we can deduce $\neg\ell_1$ to be a constant literal.

Additionally, we use a SAT-based constant check as described in [30], which is able to detect constants semantically. For this the matrix φ is passed to an incremental SAT solver and it is determined whether $\varphi \wedge \ell$ is unsatisfiable. In this case, $\neg\ell$ is constant. This method reasons only over the matrix without consideration of the dependencies and can therefore be applied without any restrictions for (D)QBF. However, ignoring the quantifiers makes this method incomplete for (D)QBF.

Another well known variable elimination technique is the detection of *equivalences*, i. e., determining whether a literal ℓ_1 is logically equivalent to another literal ℓ_2. In this case, one of the variables can be eliminated by replacing all occurrences with the other one. In the (D)QBF case one has to take into account the quantifiers and the dependencies of the affected variables. A very efficient syntactic check to detect equivalent literals is to represent all binary clauses as a directed graph and to determine the strongly connected components (SCCs) within this graph. Every literal which is contained in such an SCC is equivalent to all other literals in the same component [9]. We refer to [40] for details.

The basic (syntactic) detection and propagation of constant and pure literals as well as equivalent literals can be (and were) implemented very efficiently and turned out to be a necessary feature to let preprocessing scale. Hence, in our implementation we apply these three methods[1] after each and every more complex technique until a fixed-point is reached. This is referred to as the `simplify()` method in Algorithm 1.

After eliminating unit, pure, and equivalent literals, we start the main preprocessing loop by applying *gate substitution* [10]. To do so, we first identify definitions of logical gates within the formula, which can, e. g., result from applying Tseitin transformation [39] to a circuit (see Algorithm 1, line 4). In particular, we seek for AND gates with an arbitrary number of inputs and 2-input XOR gates [29]. For both we allow arbitrary negations on both inputs and output.[2] As many (D)QBF instances result from applications with circuits, the number of detectable gates can be very large. Once a gate definition is found, the variable y_{out}, defining the gate output, is no longer needed. Instead y_{out} is replaced in each and every clause by its definition. The defining clauses can be deleted afterwards. This direct substitution often produces smaller formulas than eliminating y_{out} by resolution (see next paragraphs), but can nevertheless produce very large formulas in some cases. Hence, it is only performed if the formula does not grow above a user-given bound. It is important that this technique is applied early, since many other methods, in particular the clause elimination methods (see next section),

[1] The syntactic constant detection using transitive implication chains is *not* included.
[2] Note, this covers also OR gates with arbitrarily negated inputs and output.

might eliminate gate defining clauses. To overcome this issue we optionally apply the concept of frozen variables and clauses [23] for gate definitions, i. e., these variables and clauses are excluded from elimination methods (with the exception of unit, pure, and equivalent literal detection).

Lastly, there are techniques for the elimination of existential and universal variables applying resolution and universal expansion, respectively. For both, the QBF generalization can be found in [5], and the DQBF version in [40]. Generally speaking, both methods eliminate a variable at the cost of expanding the formula.

Variable elimination by resolution can be applied for any existential variable y depending on all universal variables. In this case, we obtain an equisatisfiable formula by adding all possible resolvents with the pivot y and removing all clauses containing y or $\neg y$. The resolution of a variable is performed if the estimated size of the formula does not grow beyond a threshold (which is usually set to zero, i. e., resolution is only performed if the formula does not grow).

We observed a special case of resolution which is efficiently identified and always leads to a smaller formula. Therefore, we perform these resolutions more frequently – namely during BCP and blocked clause elimination (see clause elimination routines). If an existential literal $\ell^\exists$ only occurs in exactly one binary clause $\{\ell^\exists, \ell\}$ ($\neg\ell^\exists$ can occur arbitrarily often), then resolution of the pivot literal $\ell^\exists$ yields resolvents in which $\neg\ell^\exists$ is replaced by ℓ w. r. t. the original clauses. In our implementation we simply remove the clause $\{\ell^\exists, \ell\}$ and replace $\neg\ell^\exists$ with ℓ in every clause. This procedure is sound as long as $\mathrm{var}(\ell)$ is also existential and $D_{\mathrm{var}(\ell)} \subseteq D_{\mathrm{var}(\ell^\exists)}$ or $\mathrm{var}(\ell)$ is universal and $\mathrm{var}(\ell^\exists)$ depends on it.

Universal expansion [8,40] of a universal variable x allows to remove x by introducing a copy y' for every existential variable y depending on x, which has to depend on the same variables as y. Therefore every clause in which y occurs has to be copied, too, such that y is replaced by y' in the copy. Every occurrence of x in the original part of the formula is now replaced by $\top$, and every occurrence in the copied part is replaced by $\bot$ (or vice versa) resulting in an equisatisfiable formula. In our DQBF benchmark set, the number of depending existential variables is often very large and therefore we obtain a huge blow-up of the formula. Hence, in our implementation we do not apply universal expansion for DQBF. In contrast, in QBF many benchmark classes have quite small universal quantifier blocks. In this case, it turns out that the elimination of a complete universal block is often very beneficial, whereas expansion of single variables in large blocks does have a rather small impact. Therefore, we try to expand blocks with small sizes (< 20). We always try to expand the whole block as long as the blow-up of the formula is at most 50 % per variable. After each expansion step we also apply variable elimination by resolution in order to reduce the potential number of copied existential variables in the next steps as suggested in [8].

Clause Elimination Routines. As clause elimination routines [18] we understand techniques which eliminate a clause c such that deleting c yields an equisatisfiable formula.

The simplest form of clause elimination is *tautology elimination (TE)*: A clause $c \in \varphi$ is a tautology iff c contains both the literals ℓ and $\neg\ell$. Tautological clauses can be eliminated from φ. This condition is independent from the quantifier and hence can be applied for QBF and DQBF without any restrictions.

Another well-known technique is *subsumption elimination (SE)* [5]. A clause $c \in \varphi$ is subsumed if there exists another clause $c' \in \varphi$ such that the set of occurring literals in c' are a subset of those in c, i. e., if $\exists c' \in \varphi : c' \subseteq c$. In this case, c can be removed from φ. This technique is applied whenever new clauses are added to the formula and for each clause which was strengthened (see next section). Subsumption can be applied without any restrictions in the same manner for QBF as for DQBF as it yields a logically equivalent matrix.

Recently, *blocked clause elimination (BCE)* [21] has been intensively investigated. It was generalized to QBF in [6] and to DQBF in [40]. A clause $c \in \varphi$ is *blocked* if there is an existential literal $\ell^\exists \in c$ such that *every* resolvent with the pivot literal $\ell^\exists$ and the clause c is a tautology and the variable v which is responsible for the resolvent being a tautology is either universal and $\mathrm{var}(\ell^\exists)$ depends on v or v is existential and v's dependencies are a subset of $\mathrm{var}(\ell^\exists)$'s dependencies (in the QBF context this means that v is left of $\mathrm{var}(\ell^\exists)$ in the quantifier prefix). Such a blocked clause can be removed from φ without changing satisfiability. See the given literature for further details.

Furthermore, all clause elimination routines can be extended by adding so-called *hidden* and *covered* literals [18]. Simply speaking, these methods identify literals which can be added to c without changing satisfiability. These literals are added temporarily to c, obtaining a clause c'. TE, SE and BCE can be applied to c', resulting in *hidden/covered tautology/subsumption/blocked clause elimination* [18]. In case the checks were unsuccessful, the additional literals are removed. The intuition behind this literal addition is the following: The more literals a clause c contains, the more likely c is either a tautology, subsumed, or blocked. These methods are generalized to (D)QBF in [6,40], except for TE and SE with hidden/covered literals to DQBF. It is rather easy to see that these methods are sound, too; therefore we do not state an explicit proof here.

In our implementation we perform all clause elimination routines in a loop until a fixed-point is reached, i. e., until no further changes to the formula can be made (see Algorithm 1, line 7). To do so, we keep a queue of clause candidates, which are updated after removing a clause from the formula. Whenever a clause $c = \{\ell_1, \ldots, \ell_n\}$ has been removed, every clause in which at least one of the literals $\ell_1, \neg\ell_1, \ldots, \ell_n, \neg\ell_n$ occurs becomes a new candidate to be removed by one of the above methods.

Clause Strengthening Routines. Clause strengthening routines try to eliminate literals from a clause while preserving the truth value of the formula. We identify two main ways to do so.

Universal reduction (QBF [5], DQBF [1]) removes a universal literal $\ell^\forall$ from a clause $c \in \varphi$ if there are no existential literals $\ell^\exists$ in c that depend on $\ell^\forall$. In our

implementation universal reduction is applied for every added clause as well as for every clause that was strengthened by self-subsuming resolution.

Self-subsuming resolution [10] identifies two clauses c_1 and c_2 with $\ell \in c_1$, $\neg\ell \in c_2$ and $c_2 \setminus \{\neg\ell\} \subseteq c_1 \setminus \{\ell\}$, i. e., c_2 "almost subsumes" c_1 with the exception of exactly one literal ℓ, which is contained in the opposite polarity. Resolution of c_1 and c_2 with the pivot literal ℓ leads to $c_\mathrm{r} = c_1 \setminus \{\ell\}$. By adding c_r to the formula, c_1 is "self-subsumed" by c_r; therefore c_1 can be removed after this addition. Our implementation simply removes ℓ from c_1, which has the same effect. This technique leads to a logically equivalent matrix and is therefore independent of the quantification type and the dependencies of the variables; hence it can be applied to QBF and DQBF without any restrictions.

In our implementation, we iterate over all clauses in order to identify such self-subsumptions until a fixed-point is reached. To do so efficiently, we keep a queue of candidates that is updated after deleting a literal. Whenever a literal ℓ_i is removed from a clause $c = \{\ell_1, \ldots, \ell_n\}$, each clause containing at least one of $\neg\ell_1, \ldots, \neg\ell_{i-1}, \neg\ell_{i+1}, \ldots, \neg\ell_n$ is potentially self-subsuming with c.

Other Formula Modifications. As formula modifications we consider techniques which do not eliminate variables, literals or clauses, but which are able to identify properties that are helpful to decide the formula.

Whenever substituting a gate's output variable y_out with its definition is too costly, we apply *gate rewriting* [17] instead. It adds a new existential variable y'_out to the same quantifier block as y_out. For one implication direction of the Tseitin encoding of the gate, y_out is replaced by y'_out, thus delivering a double *Plaisted encoding* [31], and the occurrences of $\neg y_\mathrm{out}$ in the (D)QBF are replaced by $\neg y'_\mathrm{out}$. The purpose of this transformation is to favor detection of pure literals when the clauses including y_out evaluate to true and to increase the chance that clauses are blocked [6].

Dependency schemes [34] allow to identify dependencies of existential variables y on universal ones x as pseudo-dependencies. The dependencies are syntactically given by the order of the variables in prefix for QBFs and by the dependency sets for DQBFs. A dependency (x, y) is a pseudo-dependency, if it can be added or removed without altering the truth value of the formula. Since deciding whether a dependency is a pseudo-dependency is as hard as solving the formula itself [34,41], different sufficient criteria have been proposed, which are called dependency schemes.

During universal expansion (see variable elimination routines), we utilize the reflexive quadrangle resolution path dependency scheme [37,41], which is currently the most effective dependency scheme that is sound for both QBF and DQBF. Before expanding a universal variable x, we identify its pseudo-dependencies. All pseudo-dependencies of x do not have to be copied and neither have the clauses to be doubled in which only pseudo-dependencies and variables independent of x occur. This often leads to significantly smaller formulas after the expansion.

Lastly, we also apply SAT checks over the matrix in order to find *trivially (un)satisfied formulas.* A (D)QBF is trivially unsatisfied if the matrix φ is already

unsatisfied for an arbitrary assignment of the universal variables. For this check, we use an assignment of the universal variables which satisfies the fewest clauses, i. e., we assign x to $\top$ if x occurs in fewer clauses than $\neg x$. A (D)QBF is trivially satisfied if, after removing each occurrence of a universal literal within the matrix φ, the resulting matrix φ' is satisfiable. Finally, if a formula does not contain any universal variables after universal expansion, we immediately employ a SAT solver for deciding the resulting formula.

2.3 Implementation Details

Our tool was implemented in C++ on a 64 bits Linux machine. We can handle the standard qdimacs and dqdimacs file formats and also provide a clause interface. We are able to convert each QBF into a DQBF and vice-versa in case the dependencies of the DQBF can be linearized into a QBF prefix.

We apply all described techniques in our preprocessor within a main loop until a fixed-point is reached, i. e., no further changes in the formula arise during the latest iteration. Some (costly) techniques, like `trivialMatrixChecks()`, which use a SAT solver, are applied only once. For all SAT-based techniques we use the SAT solver Antom [36]. Whenever a routine was able to decide the (D)QBF, we immediately exit the loop and return the result.

For an efficient access to all clauses in which a literal ℓ occurs, we keep complete occurrence lists for each literal. Furthermore, we redundantly hold for each literal ℓ a list of all binary clauses in which ℓ occurs, since many of our syntactic methods, such as gate and equivalence detection, employ binary clauses.

We re-use unused variable IDs, i. e., whenever a variable was removed, we mark the index as "open" and such that it can be re-used. This avoids very large variable IDs and gaps in the data structure, which is crucial during universal expansion where many existential variables are newly introduced as a copy.

We tested different data structures for clauses. Structures based on `std::set` have the advantage of sorted ranges, which is beneficial for, e. g., subsumption and hidden/covered literal addition, but comes with the downside of more expensive access and insertion costs. On the other hand, a `std::vector` has constant access time, but checking the occurence of a literal in a clause gets more expensive. To overcome this issue we implemented a data structure which marks already occuring literals in the current clause. This "seen" data structure is also implemented as a `std::vector` with the length of the maximal literal ID. By doing so, we have efficient access on clause data, and checking whether a literal occurs in the clause becomes very cheap. By using this structure, we have measured a speed-up compared to `std::set`-based clauses of up to a factor of 4.

3 Experimental Evaluation

3.1 QBF Instances

Setting. We evaluated the effectiveness of HQSpre by comparing it against Bloqqer (Version 037) [6] and sQueezeBF [17] (we used QuBE 7.2, which

Table 1. Number of QBF instances decided by different preprocessors.

	#sat	#unsat	#fails	Time (s)
sQueezeBF	73	64	53	47548.7
Bloqqer	177	171	44	41872.0
HQSpre$_g$	162	209	29	25660.6
HQSpre	236	242	31	32127.0

includes sQueezeBF) regarding both the reduction of the input formula and the impact on several back-end solvers. Our new tool was run in two settings, its default one (HQSpre) and HQSpre$_g$, which preserves gate information. In Bloqqer and sQueezeBF two subsets of the techniques available in HQSpre are implemented, for more details the interested reader is referred to [6] and [17], respectively.

We used the testset selected for the latest QBF Evaluation (QBFEval 2016 [32]), consisting of 825 formulas. We selected several state-of-the-art QBF solvers (AIGSolve [29], AQuA-F2V [32] Caqe v2 [38], DepQBF v5.01 [25], GhostQ [19], Qesto v1.0 [20], and RAReQS v1.1 [19]) and observed the effects of HQSpre and the other preprocessors on the solvers, which are based on different solving techniques. AIGSolve was also run in a modified version named AIG-HQS, where the built-in preprocessor was replaced with HQSpre. This way, we can better evaluate how the preprocessors affect circuit-based solvers. The experiments were run on DALCO computing nodes, each having 2×8 Intel E5-2650v2 cores running at 2.6 GHz and providing 64 GB RAM. Each job[3] was run on a single core and limited to 600 s CPU time and 4 GB RAM. An overall consistency check reported no deviation in the results of different tools.

Comparing Pure Preprocessors. In Table 1 we evaluate the ability of the preprocessors to act as incomplete solvers and their efficiency. For each preprocessor under analysis, the number of formulas evaluated to true, to false, and of those on which the preprocessor fails are given; additionally, we specify the accumulated computation time needed to handle the testset. Whenever HQSpre failed, the reason was the time limit; memory consumption was not an issue for our preprocessor. HQSpre is the tool that solves the largest number of formulas and takes the least time on average to perform its transformations. HQSpre$_g$ is the fastest tool as it applies clause elimination techniques only to those clauses that do not encode gate information. Additionally, it restricts variable elimination by resolution to variables that are not gate outputs.

Formula Reduction. In Table 2, we show the main features of the formulas in the testset we used, and their counterparts after the transformation with the preprocessors under analysis. The average number of existential variables,

[3] A job consists of preprocessing and, where applicable, solving one formula. To guarantee repeatability, the sub-job of preprocessing a formula was performed once for all the solvers.

universal variables, overall variables, clauses, and quantifier alternations are given in columns 2 to 6. The testsets obtained by using the tools under analysis are split into three sub-rows ("r", "s", and "f", resp.) to distinguish between reduced, solved, and failed instances, respectively. For the reduced instances we report the averages of the quantity shown in the header before and after preprocessing, whereas for the others we report the averages of the original formulas.

The number of remaining clauses and variables for HQSpre are larger on average than for the competing preprocessors. This is mainly due to more aggressive universal expansion in HQSpre leading to many copied existential variables and clauses. On the other hand, many instances can be decided only due to this aggressive expansion. This can also be seen from the average number of universal variables: for the solved formulas, the number of universal variables is significantly smaller for HQSpre than for the other preprocessors.

At the bottom of Table 2, we also compare the numbers for all 233 instances which are neither solved nor failed for *all* applied preprocessors. Since the set of instances which are neither solved nor failed is different for each solver, this allows a better comparison of the size of the remaining formula. Also from this point of view, Bloqqer leads on average to the smallest formulas, but the difference to our preprocessor is not as large since many huge formulas for which Bloqqer fails but HQSpre does not are excluded from this presentation. As our next experiments show, the remaining larger average size does not worsen the results when applying a QBF solver to the preprocessed formula.

Combination with QBF Solvers. In Table 3, we show the overall performance of the solvers when considering our testset in its original form and when transformed by the preprocessors under analysis. For each testset, we list the number of formulas to be solved, which excludes those already solved by the preprocessor and those where the preprocessor failed. For each solver and testset, the number of solved instances includes those already solved by the preprocessor.

At first glance, we notice that HQSpre improves the state-of-the-art: for each solver, the number of solved instances is strictly higher compared to sQueezeBF and Bloqqer. CEGAR-based solvers (Caqe, RAReQS, and Qesto) take the greatest advantage from using HQSpre compared to Bloqqer, whereas search-based ones (AQuA and DepQBF) improve by a rather small number of solved instances. AIGSolve is the only solver which does not always take advantage from preprocessing: in most cases, its performance gets even degraded. This is mainly due to the underlying data structure of AIGSolve: it uses AND-Inverter graphs (AIGs), which are basically a circuit representation. Since AIGSolve applies syntactic gate detection on the clauses, any preprocessing step destroying this structure is harmful to the solver. AIG-HQS benefits most from our HQSpre$_g$ variant, where gate defining clauses and variables are untouched, and both variants of AIGSolve simply work worse if coupled with general purpose CNF preprocessors. AIGSolve contains an integrated preprocessor, which is well optimized to the AIG-based back-end solver. Still, by using HQSpre as additional front-end preprocessor, the number of solved instances increases. For the other preprocessors, results get worse because they destroy the gate

Table 2. Formula changes by preprocessing QBF: For each preprocessor, data is shown as "before → after" for just reduced formulas ("r"); for solved ("s") and failed ("f") instances we show their original size. At the bottom, the averages concern the subset made of the 233 instances all the preprocessors strictly reduced.

		∃-Vars	∀-Vars	Vars	Clauses	Q-alt.
Original		23769	570	24339	85984	17.0
sQueezeBF	r	8748 → 3674	303 → 261	9051 → 3935	40907 → 25782	17.2 → 10.2
	s	77096	259	77355	167100	20.7
	f	65893	4573	70466	416385	4.6
Bloqqer	r	8729 → 3256	608 → 548	9336 → 3805	50015 → 28933	13.4 → 6.6
	s	25938	142	26080	55860	19.9
	f	154630	3584	158214	678207	28.9
HQSpre$_g$	r	9194 → 12027	943 → 889	10137 → 12916	37949 → 68601	12.9 → 8.8
	s	28303	104	28407	91853	22.4
	f	179367	1066	180433	714867	6.7
HQSpre	r	12775 → 13232	1317 → 1232	14092 → 14463	54693 → 99763	11.1 → 7.1
	s	25797	109	25905	81852	21.4
	f	104575	77	104652	468670	8.1
Original		7037	661	7698	29498	11.3
sQueezeBF		3695	566	4226	13889	7.4
Bloqqer		2389	584	2973	13628	6.7
HQSpre$_g$		15354	634	15988	89002	7.1
HQSpre		10572	619	11191	77491	7.0

information that AIGSolve can exploit. Note, that even though our preprocessor runs until a fixed-point is reached, a second independent run can change the results, since some methods are only applied at the very beginning and not in every pass through the main preprocessing loop.

Impact on QBF Solvers. In Table 4, we show the impact of the preprocessors on the solvers regarding their robustness. For each pair, we report as a negative number (left) the amount of formulas a solver is able to solve only without preprocessing, and as a positive number (right) the amount of those instances where the preprocessor is necessary for the solver to solve them. Large positive numbers show complementarity, negative numbers close to zero demonstrate good robustness. As a solver based on a circuit representation of the formula, AIGSolve shows the highest complementarity, whereas our gate-preserving preprocessor version HQSpre$_g$ is the most robust one for this solver. For most solvers, HQSpre is the most robust preprocessor; exceptions are DepQBF, Caqe, and RAReQS whose techniques are less impaired by sQueezeBF.

3.2 DQBF Instances

Setting. We apply our preprocessor to DQBF benchmarks and use it as a front-end for the only two currently available solvers: HQS [15] and iDQ [13]. HQS is – like

Table 3. Overall results using the original QBF instances, preprocessed by BLOQQER, sQUEEZEBF, and HQSPRE. We give the number of solved instances together with the accumulated computation times in seconds. Best results for each tool are **highlighted**.

Solver	Original		sQUEEZEBF		BLOQQER		HQSPRE_g		HQSPRE	
	#	Time (s)	#	Time (s)	#	Time (s)	#	Time (s)	#	Time (s)
AQUA	330	306288	496	208396	574	163106	463	222542	**592**	**149602**
DEPQBF	434	243677	520	196531	585	157570	509	196605	**600**	**148303**
AIGSOLVE	532	188046	480	212830	518	194297	**559**	**171348**	544	175234
AIG-HQS	507	203696	440	239303	499	206756	**560**	**172549**	536	180341
CAQE	358	290370	534	195257	576	169814	485	213024	**637**	**132017**
RAREQS	337	300562	517	204385	615	144281	458	227326	**638**	**127443**
QESTO	360	291301	550	184821	606	148490	477	217943	**652**	**122782**

Table 4. Positive and negative effect of preprocessing on QBF solvers. Best results for each QBF solver are **highlighted**.

	sQUEEZEBF		BLOQQER		HQSPRE_g		HQSPRE	
AQUA	-8	+174	-11	+255	-7	+140	**-6**	**+268**
DEPQBF	**-9**	+95	-15	+166	-21	+96	-19	**+185**
AIGSOLVE	-86	+34	-75	+61	**-36**	+63	-64	**+76**
AIG-HQS	-103	+36	-70	+62	**-32**	+85	-68	**+97**
CAQE	**-8**	+184	-13	+231	-11	+138	-10	**+289**
RAREQS	**-9**	+189	-16	+294	-12	+133	-12	**+313**
QESTO	-6	+196	-14	+260	-5	+122	**-3**	**+295**

AIGSOLVE – an elimination-based solver using AIGs; IDQ is an instantiation-based approach using a SAT solver as back-end. Since HQSPRE is the first available preprocessor for DQBF, there are no competitors to compare with. We also apply the gate preserving version HQSPRE_g for this test set. Since there is no standard benchmark set for DQBF we randomly selected 499 benchmarks of different size and difficulty from currently available benchmark sets: They encompass equivalence checking problems for incomplete circuits [15,12,13], and formulas resulting from the synthesis of safe controllers [7]. We used the DALCO computing nodes with the same limitations as in our QBF experiments.

Comparing Pure Preprocessors. Table 5 shows the ability of HQSPRE to act as an incomplete solver. Since no universal expansion is applied, on the one hand HQSPRE solved fewer instances compared to the QBF benchmarks set. On the other hand, HQSPRE could preprocess all instances within the given limits.

Table 5. Decided instances of different preprocessors for DQBF.

	#sat	#unsat	#fails	Time (s)
HQSPRE_g	7	60	0	29441.2
HQSPRE	5	71	0	162615.8

Table 6. Formula shrinking after preprocessing DQBF: For each preprocessing setting, data is shown as "before → after" for just reduced ("r") formulas, and for solved ("s") formulas we show their original size. At the bottom, the averages concern the subset made of the 407 instances all the preprocessors successfully reduce but not solve.

		∃-Vars	∀-Vars	Vars	Clauses	Deps
Original		367.7	70.5	438.3	1165.1	8425.8
HQSpre$_g$	r	390.3 → 243.3	78.3 → 77.2	468.6 → 320.5	1187.9 → 883.7	9038.3 → 37.7
	s	222.3	20.4	242.7	1018.2	4476.7
HQSpre	r	402.4 → 96.1	79.6 → 78.2	482.1 → 174.3	1229.9 → 540.7	9540.8 → 589.8
	s	174.5	19.9	194.4	804.1	2219.8
Original		407.1	81.9	489.0	1240.0	9544.8
HQSpre$_g$		257.4	81.0	338.4	933.8	39.8
HQSpre		97.2	80.6	177.8	543.6	612.4

Formula Reduction. Table 6 shows the effect on the formula size for the DQBF instances in the same manner as in Table 2. Note, there are no quantifier blocks for DQBF, hence we cannot give the number of quantifier alternations. Instead, we state the number of dependencies ("deps"), which is the sum of the cardinalities of the dependency sets of the existential variables. The given number is the average over all concerned benchmarks. In the last rows, we state the numbers for the 407 commonly reduced, but not solved benchmarks.

Especially, the number of dependencies is significantly reduced for both variations. Since we do not apply any universal expansion the number of universal variables is almost unchanged – the small decrease is mainly caused by pure literal detection of universal variables. On the other hand, this strictly leads to smaller formulas in terms of variables, clauses, and dependencies. Notably, there are 18 instances with HQSpre$_g$ and 20 instances with HQSpre, respectively, for which the DQBF dependencies were linearized, i. e., the tools were able to convert the formula into an easier to solve QBF problem.

Impact on DQBF Solvers. Finally, we passed the preprocessed formulas to the two DQBF solvers and compare them with the results for the original formula. For HQS we use two versions: the usual one (HQS) and a version where we have integrated HQSpre$_g$ into the solver (HQS^I). This means that in the combination of HQSpre and HQSpre$_g$ with HQS^I the formula is actually preprocessed twice. The results are given in Table 7. As it can be seen, iDQ and HQS both significantly benefit from preprocessing. However, preprocessing the formula and feeding it into HQS in CNF form does not yield an optimal behavior of the solver compared to a tight integration as in HQS^I. The reason for this is that HQS does not apply gate detection on its own, which leads to much larger AIGs with more variables. Still, we can see that HQSpre is effective: without preprocessing, only 223 instances are solved, with gate-preserving preprocessing, but without exploiting the gate information 326 instances, and with full preprocessing 351 instances are solved. However, the best results are obtained if we integrate the preprocessor into the solver such that the gate information extracted from the

Table 7. Overall results using the original DQBF instances and preprocessed by HQSpre and HQSpre$_g$. The accumulated computation times are given in seconds. Best results for each DQBF solver are **highlighted**.

Solver	Original		HQSpre$_g$		HQSpre	
	#	Time (s)	#	Time (s)	#	Time (s)
iDQ	151	214404	170	201165	**214**	**171676**
HQS	223	176222	326	108788	**351**	**93912**
HQS^I	**456**	**30946**	450	34621	228	164299

CNF is exploited when creating the AIG data structures of the solver. In this case, 456 instances get solved. Preprocessing the formula twice as in HQSpre + HQS^I or HQSpre$_g$ + HQS^I, causes an additional overhead and modifies the formula: Since some of the more expensive techniques like SAT-based constant detection are applied only once, preprocessing the formula twice leads not only to additional overhead, but also to a different formula.

We can conclude that HQSpre is effective also for preprocessing DQBFs. For HQS as the back-end solver, it is of highest importance not only to preserve gate information, but also to integrate the preprocessor into the solver such that this information is exploited optimally.

4 Conclusion

We presented a new state-of-the-art tool HQSpre for preprocessing QBF outperforming every tested competing tool by the number of solved instances as well as increasing the number of solved instances for each state-of-the-art QBF-solver using HQSpre as front-end. Moreover our tool is able to preprocess DQBF formulas effectively and efficiently, being the first available DQBF preprocessor. An integrated version of the DQBF preprocessor clearly outperforms every other competing solver and preprocessor combination.

As future work we want to improve and enhance our gate detection methods. Namely, we want to support the Plaisted-Greenbaum encoding [31] and semantic gate detection. We like to develop an explicit gate and/or AIGER [3] interface, which also closes the gap between solver and applications in general. We also plan to expand our methodology portfolio by other well-known techniques like unit propagation look-ahead [24] (also sometimes referred to as failed literal detection) and vivification [28]. Moreover, we would like to extend our tool with Skolem and Herbrand functions in order to provide and preserve certificates. Lastly, our experimental results indicate that deciding DQBF is very efficient if we are able to transform the formula into a QBF. In order to decide whether a DQBF can be transformed into an equivalent QBF and which operations are needed to do so, a more intense utilization of dependency schemes is needed.

References

1. Balabanov, V., Chiang, H.K., Jiang, J.R.: Henkin quantifiers and Boolean formulae: A certification perspective of DQBF. Theoretical Computer Science 523, 86–100 (2014)
2. Biere, A.: Resolve and expand. In: Hoos, H.H., Mitchell, D.G. (eds.) Proc. of SAT. LNCS, vol. 3542, pp. 59–70. Springer (2004)
3. Biere, A.: Aiger format. `http://fmv.jku.at/aiger/` (2007)
4. Biere, A., Cimatti, A., Clarke, E.M., Strichman, O., Zhu, Y.: Bounded model checking. Advances in Computers 58, 117–148 (2003)
5. Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.): Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185. IOS Press (2008)
6. Biere, A., Lonsing, F., Seidl, M.: Blocked clause elimination for QBF. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) Proc. of CADE. LNCS, vol. 6803, pp. 101–115. Springer (2011)
7. Bloem, R., Könighofer, R., Seidl, M.: SAT-based synthesis methods for safety specs. In: McMillan, K.L., Rival, X. (eds.) Proc. of VMCAI. LNCS, vol. 8318, pp. 1–20. Springer (2014)
8. Bubeck, U., Kleine Büning, H.: Bounded universal expansion for preprocessing QBF. In: Marques-Silva, J., Sakallah, K.A. (eds.) Proc. of SAT. LNCS, vol. 4501, pp. 244–257. Springer (2007)
9. Del Val: Simplifying binary propositional theories into connected components twice as fast. In: Nieuwenhuis, R., Voronkov, A. (eds.) Proc. of LPAR. LNCS, vol. 2250, pp. 392–406. Springer (2001)
10. Eén, N., Biere, A.: Effective preprocessing in SAT through variable and clause elimination. In: Bacchus, F., Walsh, T. (eds.) Proc. of SAT. LNCS, vol. 3569, pp. 61–75. Springer (2005)
11. Eggersglüß, S., Drechsler, R.: A highly fault-efficient SAT-based ATPG flow. IEEE Design & Test of Computers 29(4), 63–70 (2012)
12. Finkbeiner, B., Tentrup, L.: Fast DQBF refutation. In: Sinz, C., Egly, U. (eds.) Proc. of SAT. LNCS, vol. 8561, pp. 243–251. Springer (2014)
13. Fröhlich, A., Kovásznai, G., Biere, A., Veith, H.: iDQ: Instantiation-based DQBF solving. In: Le Berre, D. (ed.) Int'l Workshop on Pragmatics of SAT (POS). EPiC Series, vol. 27, pp. 103–116. EasyChair (2014)
14. Gitina, K., Reimer, S., Sauer, M., Wimmer, R., Scholl, C., Becker, B.: Equivalence checking of partial designs using dependency quantified Boolean formulae. In: Proc. of ICCD. pp. 396–403. IEEE CS (2013)
15. Gitina, K., Wimmer, R., Reimer, S., Sauer, M., Scholl, C., Becker, B.: Solving DQBF through quantifier elimination. In: Proc. of DATE. pp. 1617–1622. IEEE (2015)
16. Giunchiglia, E., Marin, P., Narizzano, M.: QuBE7.0. Journal on Satisfiability, Boolean Modelling and Computation 7(2-3), 83–88 (2010)
17. Giunchiglia, E., Marin, P., Narizzano, M.: sQueezeBF: An effective preprocessor for QBFs based on equivalence reasoning. In: Strichman, O., Szeider, S. (eds.) Proc. of SAT. LNCS, vol. 6175, pp. 85–98. Springer (2010)
18. Heule, M., Järvisalo, M., Lonsing, F., Seidl, M., Biere, A.: Clause elimination for SAT and QSAT. Journal of Artificial Intelligence Research 53, 127–168 (2015)
19. Janota, M., Klieber, W., Marques-Silva, J., Clarke, E.M.: Solving QBF with counterexample guided refinement. In: Cimatti, A., Sebastiani, R. (eds.) Proc. of SAT. LNCS, vol. 7317, pp. 114–128. Springer (2012)

20. Janota, M., Marques-Silva, J.: Solving QBF by clause selection. In: Proc. of IJCAI. pp. 325–331. AAAI Press (2015)
21. Järvisalo, M., Biere, A., Heule, M.: Blocked clause elimination. In: Esparza, J., Majumdar, R. (eds.) Proc. of TACAS. LNCS, vol. 6015, pp. 129–144. Springer (2010)
22. Klieber, W., Sapra, S., Gao, S., Clarke, E.M.: A non-prenex, non-clausal QBF solver with game-state learning. In: Strichman, O., Szeider, S. (eds.) Proc. of SAT. LNCS, vol. 6175, pp. 128–142. Springer (2010)
23. Kupferschmid, S., Lewis, M., Schubert, T., Becker, B.: Incremental preprocessing methods for use in BMC. Formal Methods in System Design 39(2), 185–204 (2011)
24. Li, C.M., Anbulagan, A.: Heuristics based on unit propagation for satisfiability problems. In: Proc. of IJCAI Vol. 1. pp. 366–371. Morgan Kaufmann Publishers Inc. (1997)
25. Lonsing, F., Bacchus, F., Biere, A., Egly, U., Seidl, M.: Enhancing search-based QBF solving by dynamic blocked clause elimination. In: Davis, M., Fehnker, A., McIver, A., Voronkov, A. (eds.) Proc. of LPAR. LNCS, vol. 9450, pp. 418–433. Springer (2015)
26. Meyer, A.R., Stockmeyer, L.J.: Word problems requiring exponential time: Preliminary report. In: Proc. of STOC. pp. 1–9. ACM Press (1973)
27. Peterson, G., Reif, J., Azhar, S.: Lower bounds for multiplayer non-cooperative games of incomplete information. Computers & Mathematics with Applications 41(7–8), 957–992 (2001)
28. Piette, C., Hamadi, Y., Sais, L.: Vivifying propositional clausal formulae. In: Ghallab, M., Spyropoulos, C.D., Fakotakis, N., Avouris, N.M. (eds.) Proc. of ECAI. Frontiers in Artificial Intelligence and Applications, vol. 178, pp. 525–529. IOS Press (2008)
29. Pigorsch, F., Scholl, C.: Exploiting structure in an AIG based QBF solver. In: Proc. of DATE. pp. 1596–1601. IEEE (2009)
30. Pigorsch, F., Scholl, C.: An AIG-based QBF-solver using SAT for preprocessing. In: Sapatnekar, S.S. (ed.) Proc. of DAC. pp. 170–175. ACM Press (2010)
31. Plaisted, D.A., Greenbaum, S.: A structure-preserving clause form translation. Journal of Symbolic Computation 2(3), 293–304 (1986)
32. QBFEVAL'16. `http://www.qbflib.org/event_page.php?year=2016` (2016)
33. Rintanen, J., Heljanko, K., Niemelä, I.: Planning as satisfiability: parallel plans and algorithms for plan search. Artificial Intelligence 170(12-13), 1031–1080 (2006)
34. Samer, M., Szeider, S.: Backdoor sets of quantified Boolean formulas. Journal of Automated Reasoning 42(1), 77–97 (2009)
35. Scholl, C., Becker, B.: Checking equivalence for partial implementations. In: Proc. of DAC. pp. 238–243. ACM Press (2001)
36. Schubert, T., Reimer, S.: antom. In: `https://projects.informatik.uni-freiburg.de/projects/antom` (2016)
37. Slivovsky, F., Szeider, S.: Soundness of Q-resolution with dependency schemes. Theoretical Computer Science 612, 83–101 (2016)
38. Tentrup, L., Rabe, M.N.: CAQE: A certifying QBF solver. In: Proc. of FMCAD. pp. 136–143. IEEE (2015)
39. Tseitin, G.S.: On the complexity of derivation in propositional calculus. Studies in Constructive Mathematics and Mathematical Logic Part 2, 115–125 (1970)
40. Wimmer, R., Gitina, K., Nist, J., Scholl, C., Becker, B.: Preprocessing for DQBF. In: Heule, M., Weaver, S. (eds.) Proc. of SAT. LNCS, vol. 9340, pp. 173–190. Springer (2015)
41. Wimmer, R., Scholl, C., Wimmer, K., Becker, B.: Dependency schemes for DQBF. In: Proc. of SAT. LNCS, vol. 9710, pp. 473–489. Springer (2016)