Clausal Abstraction for DQBF

Leander Tentrup[1] and Markus N. Rabe[2]

[1] Reactive Systems Group, Saarland University, Saarbrücken, Germany
[2] Google Research, Mountain View, California[⋆]

Abstract. Dependency quantified Boolean formulas (DQBF) is a logic admitting existential quantification over Boolean functions, which allows us to elegantly state synthesis problems in verification such as the search for invariants, programs, or winning regions of games. In this paper, we lift the clausal abstraction algorithm for quantified Boolean formulas (QBF) to DQBF. Clausal abstraction for QBF is an abstraction refinement algorithm that operates on a sequence of abstractions that represent the different quantifier levels. For DQBF we need to generalize this principle to partial orders of abstractions. The two challenges to overcome are: (1) Clauses may contain literals with incomparable dependencies, which we address by the recently proposed proof rule called Fork Extension, and (2) existential variables may have spurious dependencies, which we prevent by tracking consistency requirements during the execution. Our implementation dCAQE solves significantly more formulas than the existing DQBF algorithms.

1 Introduction

The search for functions given declarative specifications is often called the synthesis problem and it is considered to be an extremely hard algorithmic problem. The synthesis of invariants, programs, or winning regions of (finite) games can all be expressed as the existence of a function $f\colon \mathbb{B}^m \to \mathbb{B}^n$ such that for all tuples of inputs $x_1, \dots, x_k \in \mathbb{B}^m$ some relation $\varphi(x_1, f(x_1), \dots, x_k, f(x_k))$ over function applications of f is satisfied. While it is possible to specify these problems in SMT or in first-order logic, existing algorithms struggle to solve even simple instances of synthesis queries.

In order to develop a new algorithmic approach for synthesis problems, we focus on the simplest logic admitting the existential quantification over Boolean functions, dependency quantified Boolean formulas (DQBF). However, existing algorithms for DQBF perform poorly, in particular on synthesis problems [4]. This is not surprising: Typical synthesis queries contain two or more function applications, i.e. are of the form $\exists f.\, \forall x_1, x_2.\, \varphi(x_1, f(x_1), x_2, f(x_2))$, and involve bit-vector variables, e.g. $x_1, x_2 \in \mathbb{B}^n$. The so far best performing algorithm for DQBF needs to expand either x_1 or x_2 in order to reach a linear quantifier prefix, which can then be converted to a QBF [12]. This means that they often reduce to QBF formulas that are exponential in n.

[⋆] Work partially done while at University of California, Berkeley.

Abstraction refinement algorithms have been very successful for QBF, winning the recent editions of the annual QBF competition [16, 17, 21, 23]. Inspired by this success story, we lift the abstraction refinement algorithm called *clausal abstraction* [23] to DQBF. The idea of clausal abstraction for QBF is to split the given quantified problem into a sequence of propositional problems, one for each quantifier in the quantifier prefix, and instantiate a SAT solver for each of them. The SAT solvers solve the quantified problem by communicating assignments (representing examples and counter-examples) to their neighbors.

Lifting clausal abstraction to DQBF comes with two major challenges: First, clausal abstraction is based on Q-resolution [25] and Q-resolution is sound but incomplete for DQBF [1]. In particular, clauses may contain variables from incomparable quantifiers, so called information forks [22] which characterizes the reason for incompleteness. We address this problem using the *Fork Extension* [22] proof rule, which allows us to split clauses with information fork into a set of clauses without information fork by introducing new variables. Second, clausal abstraction relies on the linear quantifier order of QBFs in prenex normal form. For DQBF, however, quantifiers can form an arbitrary partial order. When building a linear order by over-approximating the dependencies of existential variables and applying clausal abstraction naively, those variables may have *spurious dependencies*, i.e. they may only be able to satisfy all the constraints, if they depend on variables that are not allowed by the Henkin quantifiers. We show how to record consistency requirements, i.e., partial Skolem functions, that guarantee that existential variables solely depend on their stated dependencies.

In this paper, we present the first abstraction based solving approach for DQBF. The algorithm successfully applies recent insight in solving quantified Boolean formulas: It is based on the versatile and award-winning clausal abstraction framework [13, 17, 23–25] and leverages progress in DQBF proof systems [22]. Their integration in this work is non-trivial. To handle the non-linear dependencies, we use an over-approximation of the dependencies together with consistency requirements. Further, we turn clausal abstraction into an incremental algorithm that can accept new clauses and variables during solving. Our experiments show that our approach consistently outperforms first-order reasoning [9] on the DQBF benchmarks and it is especially well-suited for the synthesis benchmark set [4] where expansion-based solvers fall short.

2 Preliminaries

Let $\mathcal{V}$ be a finite set of propositional variables. We use the convention to denote universally quantified variables (short also *universals*) by x and existentially quantified variables (or *existentials*) by y. The set of all universals is denoted $\mathcal{X}$, and the set of all existentials is denoted $\mathcal{Y}$. For sets of universals and existentials we use X and Y, respectively. We consider DQBF of the form $\forall x_1. \ldots \forall x_n. \exists y_1(H_1). \ldots \exists y_m(H_m). \varphi$, that is, DQBF begin with universal quantifiers followed by *Henkin quantifiers* and the quantifier-free part φ. A Henkin quantifier $\exists y(H)$ introduces a new variable y, like a normal quantifier, but also specifies a set $H \subseteq \mathcal{X}$ of *dependencies.* A *literal* l is either a variable $v \in \mathcal{V}$ or its negation $\overline{v}$. We call the disjunction $C = (l_1 \vee l_2 \cdots \vee l_n)$ over literals a *clause*, and assume w.l.o.g. that the propositional part of DQBFs are given as a conjunction of clauses, i.e., in conjunctive normal form (CNF). We call the propositional part φ of a DQBF in CNF the *matrix* and we use C_i to refer to the ith clause of φ where unambiguous. For convenience, we treat clauses also as a set of literals and we treat matrices as a set of clauses and use the usual set operations for their manipulation. We denote by $var(l)$ the variable v corresponding to literal l. For literals l of existential variables with dependency set H we define $dep(l) = H$. For literals of universal variables we define $dep(l) = \{var(l)\}$. We lift the operator dep to clauses by defining $dep(C) = \bigcup_{l \in C} dep(l)$. We define $C|_V$ for some clause C and set of variables V as the clause $\{l \in C \mid var(l) \in V\}$.

Given a set of variables $V \subseteq \mathcal{V}$, an *assignment* of V is a function $\alpha : V \to \mathbb{B}$ that maps each variable $v \in V$ to either true ($\top$) or false ($\bot$). A *partial assignment* is a partial function from V to $\mathbb{B}$, i.e. it may be *undefined* on some inputs. To improve readability, we represent (partial) assignments also as a conjunction of literals (i.e., a cube), e.g., we write $x_1\overline{x_2}$ to denote the assignment $\{x_1 \mapsto \top, x_2 \mapsto \bot\}$. We use $\alpha \sqcup \alpha'$ as the update of partial assignment α with α', formally defined as $(\alpha \sqcup \alpha')(v) = \begin{cases} \alpha'(v) & \text{if } v \in \text{dom}(\alpha') \\ \alpha(v) & \text{otherwise} \end{cases}$.

We write $\alpha \sqsubseteq \alpha'$ if $\alpha(v) = \alpha'(v)$ for every $v \in \text{dom}(\alpha)$. To restrict the domain of an assignment α to a set of variables V, we write $\alpha|_V$. We denote *the set of assignments* of a set of variables V by $\mathcal{A}(V)$. A *Skolem function* $f_y \colon \mathcal{A}(dep(y)) \to \mathbb{B}$ maps an assignment of the dependencies of y to an assignment of y. The truth of a DQBF Φ with matrix φ is equivalent to the existence of a Skolem function f_y for every variable y of the existentially quantified variables $\mathcal{Y}$, such that substituting all existentials y in φ by their Skolem function f_y results in a valid formula. We use $\Phi[\alpha]$ to denote the replacement of variables bound by α in Φ with the corresponding value.

Relation to QBF in prenex form. In QBF the dependencies of a variable are implicitly determined by the universal variables that occur before the quantifier in the quantifier prefix. This gives rise to the notion that QBF have a *linear* quantifier prefix, whereas DQBF allows for partially ordered quantifiers.

3 Lifting Clausal Abstraction

In this section, we lift clausal abstraction to DQBF. We begin with a high level explanation of the algorithm for QBF and a discussion of the invariants that hold for QBF but are no longer valid for DQBF. For each of those we identify the underlying problem and show how we need to modify clausal abstraction. In the following subsections we then explain those extensions in detail. For the remainder of this section, we assume w.l.o.g. that we are given a DQBF Φ with matrix φ, that φ does not contain clauses with information forks, and that every clause is universally reduced. If a formula contains information forks initially, they can be removed as described in Section 3.4.

The clausal abstraction algorithm assigns existential and universal variables, where the order of assignments is determined by the quantifier prefix, until all clauses in the matrix are satisfied or there is a conflict, i.e., a set of clauses that cannot be satisfied simultaneously. Those variable assignments are generated by propositional formulas, one for every quantifier, which we call *abstractions*. In case of a conflict, the reason for this conflict is excluded by refining the abstraction at an outer quantifier.

The assignment order is based on the quantifier prefix. Thus, for QBF it holds that an existential variable is only assigned if its dependencies are assigned. In DQBF, Henkin quantifiers allow us to introduce incomparable dependency sets, and hence, in general, there is no linear order of assignments. We thus weaken this invariant by requiring that for every existential variable y, all of its dependencies have to be assigned before assigning y. We ensure this by creating a graph-based data structure, the dependency lattice, described in Section 3.1. As an immediate consequence, and in contrast to QBF, an existential variable may be assigned different values depending on assignments to non-dependencies, and we call this phenomenon a *spurious dependency*. To eliminate those spurious dependencies, we enhance the certification approach of clausal abstraction [23] to build, incrementally, a constraint system that enforces that an existential variable only depends on its dependencies. These *consistency requirements* represent partial Skolem functions. Section 3.5 describes how the consistency requirements are derived, how they are integrated in the algorithm, and when they are invalidated.

We build an abstraction for every existential quantifier $\exists Y$, splitting every clause C of the matrix into three parts, based on whether a literal $l \in C$ is (1) a dependency, (2) a literal of a variable in Y, or (3) neither of the two. Section 3.2 gives a formal description of the abstraction. As mentioned, all dependencies of Y must be assigned before we query the abstraction of the quantifier $\exists Y$ for a candidate assignment of variables Y. From the perspective of this abstraction, assignments to non-Y variables are equivalent when they satisfy the same set of clauses. Vice versa, the only information that matters for other abstractions is the set of clauses satisfied by variables Y or their dependencies. The abstraction for Y therefore defines a set of interface variables consisting of *satisfaction variables* and *assumption variables*, one for every clause C, where the satisfaction variable indicates whether the clause is satisfied by a dependency of Y and the assumption variable indicates whether C must still be satisfied by variables outside of Y. Conflicts are represented by a set of assumption variables that turned out to be not satisfiable only by variables outside of Y. Refinements are clauses over those assumption variables, requiring that at least one of those contained clauses is satisfied by an assignment to Y.

Those refinements correspond to conflict clauses in search-based algorithms and can be formalized as derived clauses in the Q-resolution calculus [25]. Since Q-resolution is incomplete for DQBF and the incompleteness can be characterized by clauses with information fork, we check if a conflict clause derived by the algorithm contains such a fork. If this is the case, we *split* this clause into

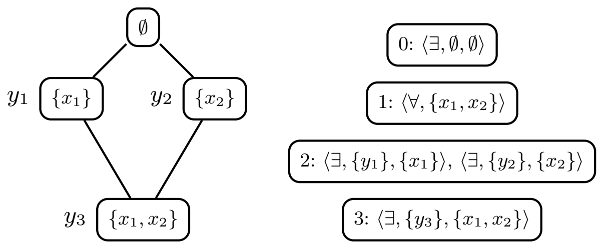
$\emptyset$
y_1 $\{x_1\}$
y_2 $\{x_2\}$
y_3 $\{x_1, x_2\}$
0: $\langle \exists, \emptyset, \emptyset \rangle$
1: $\langle \forall, \{x_1, x_2\} \rangle$
2: $\langle \exists, \{y_1\}, \{x_1\} \rangle$, $\langle \exists, \{y_2\}, \{x_2\} \rangle$
3: $\langle \exists, \{y_3\}, \{x_1, x_2\} \rangle$

Fig. 1. Dependency lattice (left) and quantifier levels (right) for the DQBF given in Example 1.

a set of clauses that are fork-free. As a byproduct, new existential variables are created. We show in Section 3.4 how clauses with information fork are split and how the clausal abstraction algorithm is extended to *incrementally* accept new clauses and variables.

Example 1. We will use the following formula with the dependency sets $\{x_1\}$, $\{x_2\}$, and $\{x_1, x_2\}$ as a running example.

$$\forall x_1, x_2 .\, \exists y_1(x_1) .\, \exists y_2(x_2) .\, \exists y_3(x_1, x_2) .$$
$$\underbrace{(x_1 \lor \overline{x_2} \lor \overline{y_2} \lor y_3)}_{C_1} \underbrace{(\overline{x_1} \lor y_2 \lor y_3)}_{C_2} \underbrace{(\overline{y_1} \lor x_2 \lor \overline{y_3})}_{C_3} \underbrace{(y_1 \lor \overline{y_3})}_{C_4} \underbrace{(x_1 \lor y_1)}_{C_5}$$

3.1 Dependency Lattice and Quantifier Levels

To lift clausal abstraction to DQBF, we need to deal with *partially* ordered dependency sets. Given a DQBF Φ, the algorithm starts with closing the dependency sets under intersection, which can also be described as building the meet-semilattice $\langle \mathcal{H}, \subseteq \rangle$. That is, $\mathcal{H}$ contains all dependency sets of variables in Φ and we add $H \cap H'$ to $\mathcal{H}$ for every $H, H' \in \mathcal{H}$ until a fixed point is reached. We call this meet-semilattice the *dependency lattice*. For our running example, we have to add the empty dependency set, resulting in the dependency lattice depicted on the left of Fig. 1. In addition to the dependency sets $\mathcal{H}$ and the edge relation $\subseteq$, we depict the existential variables next to their dependency sets.

Quantifier Levels and Nodes. We continue with building the data structure on which the algorithm operates. A *node* binds a variable of the DQBF. A universal node $\langle \forall, X \rangle$ binds universal variables X and an existential node $\langle \exists, Y, H \rangle$ binds existential variables Y with dependency set H. Nodes are grouped together in *quantifier levels*, where each universal level contains exactly one universal node and existential levels may contain multiple existential nodes. We index levels by natural numbers i, starting with 0. On the right of Fig. 1 is an example for the data structure obtained from the dependency lattice on its left. Before describing the construction of quantifier levels, we state their invariants. For some node N, let $\mathit{bound}_\forall(N)$ be the set of variables bound at N, i.e., the union of all X where $\langle \forall, X \rangle$ is in a level with smaller index than node N. Let $\mathit{bound}_\exists(N)$ be the analogously defined set of bound existential variables. The set of bound variables is $\mathit{bound}(N) \coloneqq \mathit{bound}_\exists(N) \mathbin{\dot\cup} \mathit{bound}_\forall(N)$.

Proposition 1. *The quantifier levels data structure has the following properties.*

1. *Every variable is bound excactly once, i.e., for every variable v in Φ, there is exactly one node $\langle \forall, X \rangle$ or $\langle \exists, Y, H \rangle$ such that $v \in X$ or $v \in Y$.*
2. *Every pair of nodes $\langle \exists, Y, H \rangle$ and $\langle \exists, Y', H' \rangle$ with $Y \neq Y'$ contained in an existential level have incomparable dependencies, i.e., $H \not\subseteq H'$ and $H \not\supseteq H'$.*
3. *For every pair of nodes $\langle \exists, Y_i, H_i \rangle$ and $\langle \exists, Y_j, H_j \rangle$ contained in existential levels i and j with $i < j$, it holds that either $H_i \subset H_j$ or H_i and H_j are incomparable.*
4. *For every existential node $\langle \exists, Y, H \rangle$ it holds that $H \subseteq bound_\forall(\langle \exists, Y, H \rangle)$.*
5. *There is a unique maximal $\langle \exists, Y, H \rangle$ with $H \supset H'$ for every other $\langle \exists, Y', H' \rangle$.*

In the following, we describe the construction of quantifier levels from a dependency lattice. Every element of the dependency lattice $H \in \mathcal{H}$ makes one existential node, $\langle \exists, Y, H \rangle$, where Y is the set of existential variables with dependency set H, i.e. $dep(y) = H$ for all $y \in Y$. Some existential nodes (like the root node in our example) may thus be initially empty. The existential levels are obtained by an antichain decomposition of the dependency lattice (satisfying Proposition 1.2 and 1.3). If the dependency lattice does not contain a unique maximal element, we add an empty existential node $\langle \exists, \mathcal{X}, \emptyset \rangle$ (Proposition 1.5).

Universal variables are placed in the universal node just before the existential level they first appear in as a dependency. This is achieved by a top-down pass through the existential quantifier levels, adding a universal level with node $N = \langle \forall, X \rangle$ before existential level with nodes $\langle \exists, Y_1, H_1 \rangle, \ldots, \langle \exists, Y_k, H_k \rangle$ such that $X = \left(\bigcup_{1 \leq i \leq k} H_i \right) \setminus bound_\forall(N)$ (Proposition 1.4). Empty universal levels $\langle \forall, \emptyset \rangle$ are omitted. Level numbers follow the inverse order of the dependency sets, such that the "outer" quantifiers have smaller level numbers than the "inner" quantifiers; see Fig. 1.

If the formula is a QBF, it holds that $bound_\forall(\langle \exists, Y, H \rangle) = H$. For QBF, this construction yields a strict alternation between universal and existential levels, but for DQBF existential levels can succeed each other, as shown in Fig. 1.

Algorithmic Overview. The overall approach of the algorithm is to construct a propositional formula θ for every node, that represents which clauses it can satisfy (for existential nodes) or falsify (for universal nodes). We describe their initialization in detail below. In every iteration of the loop in algorithm SOLVE (Fig. 2) the variable assignment α_V is extended (case CandidateFound), which we assume to be globally accessible, or node abstractions are refined by adding an additional clauses (case Conflict).

The nodes are responsible for determining candidate assignments to the variables bound at that node, or to give a reason why there is no such assignment. If a node is able to provide a candidate assignment, we proceed to the successor level (Fig. 2, line 7). A conflict occurs when the algorithm determines that the current assignment α_V definitely violates the formula (unsat conflict) or satisfies it (sat conflict). When conflicts are inspected (explained in Section 3.3), they indicate a level that tells the main loop how far we have to jump back (Fig. 2,

```
1: procedure Solve(DQBF $\Phi$)
2:     $levels \leftarrow$ build quantifier levels
3:     initialize every node in $levels$, i.e., build abstraction $\theta$, set $entries \leftarrow []$
4:     $\alpha_V \leftarrow \{\}$, $lvl \leftarrow 0$
5:     loop
6:         match SolveLevel($lvl$)
7:             CandidateFound $\Rightarrow$ $lvl \leftarrow lvl + 1$
8:             Conflict($jmpBackToLvl$) $\Rightarrow$ $lvl \leftarrow jmpBackToLvl$
9:             Result($res$) $\Rightarrow$ return $res$
```

Fig. 2. Main solving algorithm that iterates over the quantifier levels.

line 8). The last alternative in the main loop is that we have found a result, which allows us to terminate (line 9).

3.2 Initialization of the abstractions θ

The formula θ for each node represents how the node's variables interact with the assignments on other levels. The algorithm guarantees that whenever we generate a candidate assignment for a node, all variables on outer (=smaller) levels have a fixed assignment, and thus some set of clauses is satisfied already. Existential nodes then try to satisfy more clauses with their assignment, while universal nodes try to find an assignment that makes it harder to satisfy all clauses. An existential variable y may not only depend on assignments of its dependencies, but also on assignments of existential variables with strict smaller dependency as they are in a strictly smaller level (see Section 3.1) and thus are assigned before y. We call this the extended dependency set, written $exdep(y)$, and it is defined as $dep(y) \mathbin{\dot\cup} \{y' \in \mathcal{Y} \mid dep(y') \subset dep(y)\}$. For a set $Y \subseteq \mathcal{Y}$, we define $exdep(Y) = \bigcup_{y \in Y} exdep(y)$.

The interaction of abstractions is established by a common set of *clause satisfaction* variables S, one variable $s_i \in S$ for every clause $C_i \in \varphi$. Given some existential node $\langle \exists, Y, H \rangle$ with extended dependency set $D = exdep(Y)$ and assignment α_V of outer variables V (w.r.t. $\exists Y$, i.e., $V = bound(\langle \exists, Y, H \rangle)$). For every clause $C_i \in \varphi$ it holds that if s_i is assigned to true, one of its dependencies has satisfied the clause, that is, $\alpha_V \vDash C_i|_D$. Thus, an assignment of the satisfaction variables α_S is an abstraction of the concrete variable assignment α_V as multiple assignments could led to the same satisfied clauses.

For universal quantifiers, this abstraction is sufficient as the universal player tries to satisfy as few clauses as possible. For existential quantifiers, however, the existential player can choose to either satisfy the clause directly or *assume* that the clause will be satisfied by an inner quantifier. Thus, we add an additional type of variables A, called *assumption variables*, with the intended semantics that a_i is set to false at some existential quantifier $\exists Y$ implies that the clause C_i is satisfied at this quantifier (either by an assignment α_Y to variables Y of the current node or an assignment of dependencies represented by an assignment α_S to the satisfaction variables S), formally, $\alpha_V \mathbin{\dot\sqcup} \alpha_Y \vDash C_i|_{D \dot\cup Y}$ if a_i is false.

```
1: procedure SolveLevel($lvl$)
2:     if $lvl$ is universal then return Solve$_\forall$($levels[lvl]$)
3:     for each node $n$ in $levels[lvl]$ do
4:         if Solve$_\exists$($n$) = Conflict($jmpBackToLvl$) then
5:             return Conflict($jmpBackToLvl$)
6:     return CandidateFound
```

Fig. 3. Algorithm for solving a quantifier level by iterating over the contained nodes.

We continue by defining the abstraction that implements this intuition. Formally, for every node $\langle\exists, Y, H\rangle$ and every clause C_i, we define $C_i^< := \{l \in C_i \mid \mathit{var}(l) \in \mathit{exdep}(Y)\}$ as the set of literals on which the current node may depend, $C_i^= := \{l \in C_i \mid \mathit{var}(l) \in Y\}$ as the the set of literals which the current node binds, and $C_i^> := \{l \in C_i \mid \mathit{var}(l) \notin \mathit{exdep}(Y) \cup Y\}$ as the set of literals on which the current node may not depend. By definition, it holds that $C = C_i^< \dot\cup C_i^= \dot\cup C_i^>$. The clausal abstraction θ_Y for this node is defined as $\bigwedge_{C_i \in \varphi} (a_i \vee s_i \vee C_i^=)$. Note, that s_i and a_i are omitted if $C_i^< = \emptyset$ and $C_i^> = \emptyset$, respectively.

Over time, the algorithm calls each node potentially many times for candidate assignments, and it adds new clauses learnt from refinements. The new clauses for existential nodes will only contain literals from assumption variables $L \subseteq A$, representing sets of clauses that together cannot be satisfied by the inner levels. The refinement $\bigvee_{a_i \in L} \overline{a_i}$ ensures that some clause C_i with $a_i \in L$ is satisfied at this node.

Universal nodes $\langle\forall, X\rangle$ have the objective to falsify clause. We define the abstraction θ_X for this node as $\bigwedge_{C_i \in \varphi} (s_i \vee \neg C_i^=) = \bigwedge_{C_i \in \varphi} \left(s_i \vee \bigwedge_{l \in C_i^=} \overline{l}\right)$. Observe that universal nodes do not have separate sets of variables A and S, but just one copy S. This is just a minor simplification, exploiting the formula structure of universal nodes. Note that s_i set to false implies that α_X falsifies the literals in the clause, that is, $\alpha_X \vDash \neg C_i^=$. Refinements are represented as clauses $\bigvee_{s_i \in L} \overline{s_i}$ over literals in S.

In our running example, clauses 3–5 $(\overline{y_1} \vee x_2 \vee \overline{y_3})(y_1 \vee \overline{y_3})(x_1 \vee y_1)$ are represented at node $\langle\exists, \{y_1\}, \{x_1\}\rangle$ by clauses $(a_3 \vee \overline{y_1})(a_4 \vee y_1)(y_1)$. Note especially, that $x_2 \notin \mathit{exdep}(y_1) = \{x_1\}$, thus there is no s variable in the first encoded clause, despite x_2 being assigned earlier in the algorithm (Fig. 1).

3.3 Solving Levels and Nodes

SolveLevel in Fig. 3 directly calls Solve$_\forall$ or Solve$_\exists$ on all the nodes in the level. For existential levels, if any node returns a conflict, the level returns that conflict (Fig. 3, line 5).

We assume a SAT solver interface $\textsc{sat}(\psi, \alpha)$ for matrices ψ and assumptions (represented by an assignment) α. It returns either $\mathsf{Sat}(\alpha')$, which means the formula is satisfiable with assignment $\alpha' \sqsupseteq \alpha$, or $\mathsf{Unsat}(\alpha')$, which means the formula is unsatisfiable and $\alpha' \sqsubseteq \alpha$ are the failed assumptions (i.e. the unsat core), that is, $\textsc{sat}(\psi, \alpha')$ is unsatisfiable as well.

We process universal and existential nodes with the two procedures shown in Fig. 4. The SAT solvers generate a candidate assignment to the variables (lines 4

```
1: procedure $\textsc{Solve}_\exists(node \equiv \langle\exists, Y, H\rangle)$
2:     $\alpha_Y \leftarrow \textsc{checkConsistency}(\alpha_V)$
3:     $\alpha_S \leftarrow prj_\exists(Y, \alpha_V)$
4:     match $\textsc{sat}(\theta_Y, \alpha_Y \sqcup \alpha_S)$
5:         Unsat($\alpha$) $\Rightarrow$
6:             return $\textsc{refine}(unsat, \alpha|_S, node)$
7:         Sat($\alpha$) $\Rightarrow$ update $\alpha_V$ with $\alpha|_Y$
8:             if node is maximal element then
9:                 return $\textsc{refine}(sat, \alpha_S, node)$
10:            return CandidateFound
```

```
11: procedure $\textsc{Solve}_\forall(node \equiv \langle\forall, X\rangle)$
12:     $\alpha_S \leftarrow prj_\forall(X, \alpha_V)$
13:     match $\textsc{sat}(\theta_X, \alpha_S)$
14:         Unsat($\alpha$) $\Rightarrow$
15:             return $\textsc{refine}(sat, \alpha|_S, node)$
16:         Sat($\alpha$) $\Rightarrow$ update $\alpha_V$ with $\alpha|_X$
17:             return CandidateFound
```

Fig. 4. Process existential and universal nodes.

and 13) of that node, which is then used to extend the (global) assignment α_V (lines 7 and 16). In case the SAT solver returns Unsat, the unsat core represents a set of clauses that cannot be satisfied (for existential nodes) or falsified (for universal nodes). The unsat core is then used to refine an outer node (lines 6 and 15) and we proceed with the level returned by REFINE.

Solving Existential Nodes. There are some differences in the handling of existential and universal nodes that we look into now. The linear ordering of the levels in our data structure means that there may be a variable assigned that an existential node must not depend on. We therefore need to *project* the assignment α_V to those variables in the node's dependency set. We define a function $prj_\exists \colon 2^{\mathcal{Y}} \times \mathcal{A}(V) \to \mathcal{A}(S)$ that maps variable assignments α_V to assignments of satisfaction variables S such that s_i is set to true if, and only if, some literal $l \in C_i^<$ is assigned positively by α_V. Thus, the projection function only considers actual dependencies of $\langle\exists, Y, H\rangle$:

$$prj_\exists(Y, \alpha_V)(s_i) = \begin{cases} \top & \text{if } \alpha_V \vDash C_i^< \\ \bot & \text{otherwise} \end{cases}$$

For our running example, at node $\langle\exists, \{y_2\}, \{x_2\}\rangle$, the projection for the first clause $C_1 = (x_1 \vee \overline{x_2} \vee \overline{y_2} \vee y_3)$ is $prj_\exists(\{y_2\}, \overline{x_1}\boldsymbol{x_2})(s_1) = prj_\exists(\{y_2\}, x_1\boldsymbol{x_2})(s_1) = \bot$ and $prj_\exists(\{y_2\}, \overline{x_1}\overline{\boldsymbol{x_2}})(s_1) = prj_\exists(\{y_2\}, x_1\overline{\boldsymbol{x_2}})(s_1) = \top$ because $C_1^< = (\overline{x_2})$.

If the SAT solver returns a candidate assignment at the maximal existential node (i.e., the node on innermost level), we know that all clauses have been satisfied, and we have therefore refuted the candidate assignment of some universal node. This is handled by calling REFINE in line 9. For existential nodes we additionally have to check for consistency, which we discuss in Section 3.5 (called in line 2).

Solving Universal Nodes. Similar to the projection for existential nodes, we need an (almost symmetric) projection for universal nodes (line 12). It has to differ slightly from $prj_\exists$, because we use just one set of variables S for universal nodes. A universal quantifier cannot falsify the clause if it is already satisfied.

$$prj_\forall(X, \alpha_V)(s_i) = \begin{cases} \top & \text{if } \alpha_V \vDash C_i|_{bound\langle\forall, X\rangle} \\ undef & \text{otherwise} \end{cases}$$

```
1: procedure REFINE($res, \alpha_S, node$)
2:     if $res = unsat$ then
3:         $C_{conflict} = \bigcup_{C_i \in \varphi, \alpha_S(s_i)=\bot} C_i|_{bound(node)}$        ▷ $C_{conflict}$ is universally reduced
4:         if $C_{conflict}$ contains information fork then
5:             fork elimination $\Rightarrow$ add clauses and variables, update abstractions $\theta$
6:             RESETCONSISTENCY for all nodes
7:             return Conflict($lvl = 0$)
8:     if $next \leftarrow$ DETERMINEREFINEMENTNODE($res, \alpha_S, node.level$) then
9:         return Conflict($next.level$)
10:    else                                                        ▷ conflict at outermost $\exists/\forall$ node
11:        return Result($res$)
```

Fig. 5. Refinement algorithm that applies Fork Extension in case of information forks.

3.4 Refinement

Algorithm REFINE in Fig. 5 is called whenever there is a conflict, i.e. whenever it is clear that α_V satisfies the formula (*sat* conflict) or violates it (*unsat* conflict). In case there is an unsat conflict at an existential node, we build the (universally reduced) conflict clause from α_S [25] in line 3. If the clause is fork-free, we can apply the standard refinement for clausal abstraction [23] with the exception that we need to find the unique refinement node first (line 8). This backward search over the quantifier levels is shown in Fig. 6. For an *unsat* conflict, we traverse the levels backwards until we find an existential node that binds a variable contained in the conflict clause. Because the conflict clause is fork-free, the target node of the traversal is unique. For a *sat* conflict, we do the same for universal nodes but the uniqueness comes from the fact that universal levels are singletons. We then add the refinement clause to the SAT solver at the corresponding node (lines 6 and 10) and proceed. For *sat* conflicts, we have to additionally learn consistency requirements at existential nodes (line 13) that make sure that the node produces the same result if the assignment (restricted to the dependencies of that node) repeats. In case the conflict propagated beyond the root node, we terminate with the given result.

Fork Extension. In case that the conflict clause contains a fork, we apply Fork Extension [22][3]. *Fork Extension* allows us to split a clause $C_1 \vee C_2$ by introducing a fresh variable y. The dependency set of y is defined as the intersection $dep(C_1) \cap dep(C_2)$ and represents that the question whether C_1 or C_2 satisfies the original clause needs to be resolved based on the information that is available to both of them. Fork Extension is usually only applied when C_1 and C_2 have incomparable dependencies ($dep(C_1) \not\subseteq dep(C_2)$ and $dep(C_1) \not\supseteq dep(C_2)$), as only then the dependency set of y is smaller than those of C_1 and of C_2. The formal definition of the rule is

$$\frac{C_1 \cup C_2 \qquad y \text{ is fresh}}{\exists y(dep(C_1) \cap dep(C_2)).\ C_1 \cup \{y\} \ \wedge\ C_2 \cup \{\overline{y}\}} \ \textbf{FEx}$$

[3] Fork Extension as introduced in [22] is incomplete for general DQBF. However, it is complete for a normal form of DQBF. We refer to the full version [26] for details.

```
 1: procedure DetermineRefinementNode($res, \alpha_S, lvl$)
 2:     while $lvl \geq 0$ do
 3:         if $res = unsat$ and $lvl$ is existential then
 4:             for node $\langle \exists, Y, H \rangle$ in $levels[lvl]$ do    ▷ check if $Y$ is contained in conflict clause
 5:                 if $C_i|_Y \neq \emptyset$ for some $C_i \in \varphi$ with $\alpha_S(s_i) = \bot$ then
 6:                     $\theta_Y \leftarrow \theta_Y \wedge \bigvee_{C_i \in \varphi, \alpha_S(s_i) = \bot} \overline{a_i}$    ▷ refine abstraction
 7:                     return $\langle \exists, Y, H \rangle$
 8:         else if $res = sat$ and $lvl$ is universal with node $\langle \forall, X \rangle$ then
 9:             if $C_i|_X \neq \emptyset$ for some $C_i \in \varphi$ with $\alpha_S(s_i) = \top$ then
10:                 $\theta_X \leftarrow \theta_X \wedge \bigvee_{C_i \in \varphi, \alpha_S(s_i) = \top} \overline{s_i}$    ▷ refine abstraction
11:                 return $\langle \forall, X \rangle$
12:         else if $res = sat$ and $lvl$ is existential then    ▷ add consistency requirements
13:             learnEntry($N$) for each $N = \langle \exists, Y, H \rangle$ in $levels[lvl]$ with $H \subset bound_\forall(N)$
14:         $lvl \leftarrow lvl - 1$
15:     return $res$
```

Fig. 6. Backward search algorithm to determine refinement node.

Example 2. As an example of applying Fork Extension, consider the quantifier prefix $\forall x_1 x_2 .\, \exists y_1(x_1).\, \exists y_2(x_2)$ and clause $(\overline{x_1} \vee y_1 \vee y_2)$. Applying **FEx** with the decomposition $C_1 = \{\overline{x_1}, y_1\}$ and $C_2 = \{y_2\}$ results in the clauses $(\overline{x_1} \vee y_1 \vee \boldsymbol{y_3})(\overline{\boldsymbol{y_3}} \vee y_2)$ where y_3 is a fresh existential variable with dependency set $dep(y_3) = \emptyset$ ($dep(C_1) = \{x_1\}$ and $dep(C_2) = \{x_2\}$).

After applying Fork Extension, we encode the newly created clauses and variables within their respective nodes. We update the abstractions with those fresh variables and clauses as for the initial abstraction discussed in Section 3.2. Additionally, we reset learned Skolem functions as they may be invalidated by the refinement (Fig. 5, line 6).

3.5 Consistency Requirements

The algorithm described so far produces correct refutations in case the DQBF is false. For positive results, the *consistency* of Skolem functions of *incomparable* existential variables may be violated. Consider for example the formula $\forall x_1 \forall x_2 .\, \exists y_1(x_1).\, \exists y_2(x_2).\, \exists y_3(x_1, x_2).\, \varphi$ and assume that for the assignment $\overline{x_1}\overline{x_2}$, there is a corresponding satisfying assignment $\overline{y_1}\overline{y_2}\overline{y_3}$. If the next assignment is $\overline{x_1}\boldsymbol{x_2}$, then the assignment to y_1 has to be the same as before ($y_1 \rightarrow \bot$) as the value of its sole dependency x_1 is unchanged.

We enhance the certification capabilities of clausal abstraction [23] to build consistency requirements that represent partial Skolem functions in our algorithm during solving. We incrementally build a list of *entries*, where the first component in an entry is a propositional formula over the dependencies and the second component is the corresponding assignment α_Y. Before generating a candidate assignment in Solve$_\exists$, we call checkConsistency (Fig. 7) to check if the assignment α_Y for the given assignment α_V of dependencies is already determined, by iterating through the learned entries (Fig. 7, lines 2–3). If it is

```
1: procedure checkConsistency($\alpha_V$)
2:     for ($cond, \alpha_Y$) in entries do
3:         if sat($cond, \alpha_V$) is Sat then return $\alpha_Y$
4:     return empty assignment
5: procedure resetConsistency
6:     entries ← []
7:     reset learned clauses at universal nodes
8: procedure learnEntry($node \equiv \langle \exists, Y, H \rangle$)
9:     let $\alpha_S$ and $\alpha_Y$ be from line 7 of Fig. 4.
10:    entries.push(($\bigwedge_{C_i | \alpha_S(s_i) = \top} C_i^<, \alpha_Y$))
```

Fig. 7. Algorithms for handling consistency requirements.

the case, we get an assignment α_Y that is then assumed for the candidate generation. Note that in this case, the SAT call in line 4 of SOLVE$_\exists$ is guaranteed to return Sat (we already verified this assignment, otherwise it would not have been learned). Further, consistency requirements are only needed for existential nodes $\langle \exists, Y, H \rangle$ with $H \subset \mathit{bound}_\forall(\langle \exists, Y, H \rangle)$, i.e., that observe an over-approximation of their dependency set. For those nodes, the consistency requirements enforce that whenever two assignments of the dependencies are equal, the assignment of α_Y returns the same value as well. We call RESETCONSISTENCY (Fig. 7) to reset the consistency requirements in case we applied Fork Extension (Fig. 5, line 6) as the new clauses may affect already learned parts of the function. We, further, have to reset the clauses learned at universal nodes (Fig. 7, line 7).

We *learn* a new consistency requirement by calling LEARNENTRY (Fig. 7) on the backward search on sat conflicts, that is in line 13 in Fig. 6. When we determine the refinement node for sat conflicts, we call LEARNENTRY in every existential node $\langle \exists, Y, H \rangle$ with $H \subset \mathit{bound}_\forall(\langle \exists, Y, H \rangle)$ on the path to that node. In our example, when the base case of $\langle \exists, \{y_3\}, \{x_1, x_2\} \rangle$ returns (all clauses are satisfied, line 9 in Fig. 4), we add consistency requirmenets at nodes $\langle \exists, \{y_2\}, \{x_2\} \rangle$ and $\langle \exists, \{y_1\}, \{x_1\} \rangle$ before refining at $\langle \forall, \{x_1, x_2\} \rangle$.

3.6 Example

We consider a possible execution of the presented algorithm on our running example. For the sake of readability, we combine unimportant steps and focus on the interesting cases. Assume the following initial assignment $\alpha_1 = x_1 \overline{x_2} \overline{y_1} \overline{y_2}$ before node $N_{max} \equiv \langle \exists, \{y_3\}, \{x_1, x_2\} \rangle$. The result of projecting function $\mathit{prj}_\exists(\{y_3\}, \alpha_1)$ is $s_1 \overline{s_2} s_3 \overline{s_4} s_5$ and the SAT solver (Fig. 4, line 4) returns $\mathsf{Unsat}(\alpha_1')$ with core $\alpha_1' = \overline{s_2} \overline{s_4}$ as there is no way to satisfy both clauses $(s_2 \vee y_3)$ and $(s_4 \vee \overline{y_3})$ of the abstraction. The refinement algorithm (Fig. 5) builds the conflict clause $C_{\mathit{conflict}} = C_2|_{\mathit{bound}(N_3)} \cup C_4|_{\mathit{bound}(N_3)} = (\overline{x_1} \vee y_2 \vee y_1)$ at line 3 which contains an information fork between y_1 and y_2. We have already seen in Example 2 that the fork can be eliminated resulting in fresh variable y_4 with $\mathit{dep}(y_4) = \emptyset$ and the clauses 6 and 7 $(\overline{x_1} \vee y_1 \vee y_4)(\overline{y_4} \vee y_2)$.

Now, the root node contains variable y_4, for which we assume assignment $\{y_4 \mapsto \top\}$. For the same universal assignment as before $(x_1 \overline{x_2})$, the assignment of y_2 has to change to $\{y_2 \mapsto \top\}$ due to the newly added clause 7, leading to

$\alpha_2 = x_1\overline{x_2}\overline{y_1}y_2y_4$ before node N_{max}. The only unsatisfied clause is C_4 which can be satisfied using $\{y_3 \mapsto \bot\}$, leading to the base case (Fig. 4, line 9). During refinement, we learn Skolem function entries $(x_1 \wedge y_4, \overline{y_1})$ and $(\overline{x_2} \wedge y_4, y_2)$ at nodes $\langle\exists, \{y_1\}, \{x_1\}\rangle$ and $\langle\exists, \{y_2\}, \{x_2\}\rangle$ as $\mathit{prj}_\exists(\{y_1\}, x_1\overline{x_2})$ and $\mathit{prj}_\exists(\{y_2\}, x_1\overline{x_2})$ assign s_1, s_5, s_6 and s_1, s_6 positively, respectively.

For the following universal assignment $\overline{x_1}\overline{x_2}$, the value of y_2 is already determined by the consistency requirements (Fig. 4, line 2) to be positive. There is a continuation of the algorithm without further unsat conflict, determining that the instance is true.

3.7 Correctness

We sketch the correctness argument for the algorithm, which relies on formal arguments regarding correctness and certification of the clausal abstraction algorithm [23] and the subsequent analysis of the underlying proof system [25].[4] For soundness, the algorithm has to guarantee that existential variables are assigned consistently, that is for an existential variable y with dependency $\mathit{dep}(y)$ it holds that $f_y(\alpha) = f_y(\alpha')$ if $\alpha|_{\mathit{dep}(y)} = \alpha'|_{\mathit{dep}(y)}$ for every α and α'. Our algorithm maintains this property at every point during the execution by a combination of over-approximation and consistency requirements. Completeness relies on the fact that the underlying proof system is refutationally complete for DQBF. Progress is guaranteed as there are only finitely many different conflict clauses and, thus, only finitely many Skolem function resets.

4 Evaluation

We compare our prototype implementation, called dCAQE[5], against the publicly available DQBF solvers, iDQ [10], HQS [12], and iProver [19]. We ran the experiments on machines with a 3.6 GHz quad-core Xeon processor with timeout and memout set to 10 minutes and 8 GB, respectively. We used the DQBF preprocessor HQSPre [28] for every solver except HQS. We evaluate our solver on the DQBF case studies regarding reactive synthesis [4] and the partial equivalence checking problem (PEC) [7, 12].

The first two benchmark sets consider the partial equivalence checking (PEC) problem [11], that is, the problem whether a circuit containing not-implemented (combinatorial) parts, so called "black boxes", can be completed such that it is equivalent to a reference circuit. The inputs to the circuit are modeled as universally quantified variables and the outputs of the black boxes as existentially quantified variables. Since the output of a black box should only depend on the inputs that are actually visible to the black box, we need to restrict the dependencies of the existentially quantified variables to subsets of the universally quantified variables. The benchmark sets PEC1 and PEC2 refers to [7]

[4] A formal correctness proof is given in the full version [26].

[5] Available at `https://github.com/ltentrup/caqe`.

Table 1. Number of instances solved within 10 min. For every solver, we give the number of solved instances overall (#) and broken down by satisfiable (⊤), unsatisfiable (⊥), and uniquely solved instances (∗).

Benchmark	#	DCAQE				IDQ				HQS				IPROVER			
		#	⊤	⊥	∗	#	⊤	⊥	∗	#	⊤	⊥	∗	#	⊤	⊥	∗
PEC1 [7]	1000	**839**	7	**832**	**224**	37	0	37	0	636	**10**	626	32	71	0	71	0
PEC2 [12]	720	342	71	271	12	214	45	169	0	**401**	**104**	**297**	**60**	288	60	228	0
BoSy [4]	1216	**1006**	**389**	**617**	**66**	924	335	589	2	735	231	504	0	946	370	576	20
	2936	**2187**				1175				1772				1305			

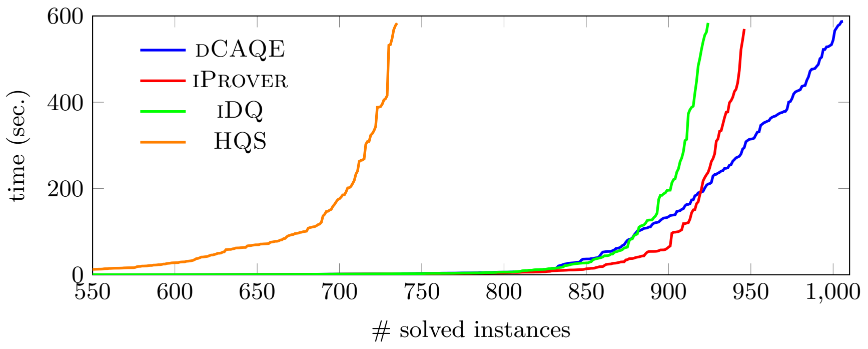

Fig. 8. Cactus plot for the BoSy benchmark.

and [12], respectively. The second case study (BoSy) considers the problem of synthesizing sequential circuits from specifications given in linear-time temporal logic (LTL) [4]. The benchmarks were created using the tool BoSy [5] and the LTL benchmarks from the Reactive Synthesis Competition [14,15]. Each formula encodes the existence of a sequential circuit that satisfies the LTL specification.

The results are presented in Table 1. The PEC instances contain over-proportionally many unsatisfiable instances and we conjecture that the differences between DCAQE and IDQ/IPROVER can be explained by the effectiveness of the resolution-based refutations that DCAQE is based on. HQS performs well on those benchmarks as well, which could be due to the fact that it implements the fast refutation technique [7] that was introduced alongside the benchmark set PEC1. The reactive synthesis benchmark set is were DCAQE excels. The benchmark set contains many easily solvable benchmarks, indicated by the high number of instances that are commonly solved by all solvers. However, there are also a fair amount of hard instances and DCAQE solves significantly more of those than any other solver. Further, we can see the effect mentioned in the introduction of infeasibility of expansion-based methods as shown by the result of HQS. The cactus plot given in Fig. 8 shows that DCAQE makes more progress, especially with a larger runtime where the other solvers solve very few instances after 100s. These results give rise to the hope that the scalability of more expressive synthesis approaches [3,6,8] can be improved by employing DQBF solving.

5 Related Work

The satisfiability problem for DQBF was shown to be NExpTime-complete [20]. Fröhlich et al. [9] proposed a first detailed solving algorithm for DQBF based on DPLL. They already encountered many challenges of lifting QBF algorithms to DQBF, like Skolem function consistency, replay of Skolem functions, forks in conflict clauses, but solved them differently. Their algorithm, called DQDPLL, has some similarities to our algorithm (in the same way that clausal abstraction and QDPLL share the same underlying proof system [25]), but performs significantly worse [9]. We highlight a few differences which we believe to be crucial: (1) Our algorithm tries to maintain as much order as possible. Placing universal nodes at the latest possible allows us to apply the cheaper QBF refinement method more often. (2) We learn consistency requirements only if they have been verified to satisfy the formula, while DQDPLL learns them on decisions. Consequently, in DQDPLL, learned Skolem functions become part of the clauses, thus, making conflict analysis more complicated and less effective as they may be undone during solving. We keep the consistency requirements distinct from the clauses, all learned clauses at existential nodes are thus valid during solving. (3) Skolem functions in DQDPLL are represented as clauses representing truth-table entries, thus, become quickly infeasible. In contrast, we use a separate certification mechanism as in QBF solvers [23]. iDQ [10] uses an instantiation-based algorithm which is based on the Inst-Gen calculus, a state-of-the art decision procedure for the effectively propositional fragment of first-order logic (EPR), which is also NExpTime-complete. HQS [12] is an expansion based solver that expands universal variables until the resulting instance has a linear prefix and applies QBF solving afterwards. Bounded unsatisfiability [7] asserts the existence of a partial (bounded) expansion tree that guarantees that no Skolem function exists. QBF preprocessing techniques have been lifted to DQBF [27, 28]. Our solving technique is based on clausal abstraction [23] (also called clause selection [17]) for QBF, which can provide certificates [23]. Later, it was shown that refutation in clausal abstraction can be simulated by Q-resolution [25].

6 Conclusions

We lifted the clausal abstraction algorithm to DQBF. This algorithm is the first to exploit the new Fork Resolution proof system and it significantly increases performance of DQBF solving on synthesis benchmarks. In particular, in the light of the past attempts to define search algorithms [9] (which are closely related to clausal abstraction) for DQBF this is a surprising success. It appears that the Fork Extension proof rule was the missing piece in the puzzle to build search/abstraction algorithms for DQBF.

Acknowledgments. We thank Bernd Finkbeiner for his valuable feedback on earlier versions of this paper. This work was partially supported by the German Research Foundation (DFG) as part of the Collaborative Research Center “Foundations of Perspicuous Software Systems” (TRR 248, 389792660) and by the European Research Council (ERC) Grant OSARES (No. 683300).

References

1. Balabanov, V., Chiang, H.K., Jiang, J.R.: Henkin quantifiers and Boolean formulae: A certification perspective of DQBF. Theor. Comput. Sci. **523**, 86–100 (2014). https://doi.org/10.1016/j.tcs.2013.12.020
2. Balabanov, V., Jiang, J.R.: Unified QBF certification and its applications. Formal Methods in System Design **41**(1), 45–65 (2012). https://doi.org/10.1007/s10703-012-0152-6
3. Coenen, N., Finkbeiner, B., Sánchez, C., Tentrup, L.: Verifying hyperliveness. In: To appear in the proceedings of CAV (2019)
4. Faymonville, P., Finkbeiner, B., Rabe, M.N., Tentrup, L.: Encodings of bounded synthesis. In: Proceedings of TACAS. LNCS, vol. 10205, pp. 354–370 (2017). https://doi.org/10.1007/978-3-662-54577-5_20
5. Faymonville, P., Finkbeiner, B., Tentrup, L.: BoSy: An experimentation framework for bounded synthesis. In: Proceedings of CAV. LNCS, vol. 10427, pp. 325–332. Springer (2017). https://doi.org/10.1007/978-3-319-63390-9_17
6. Finkbeiner, B., Hahn, C., Lukert, P., Stenger, M., Tentrup, L.: Synthesizing reactive systems from hyperproperties. In: Proceedings of CAV. LNCS, vol. 10981, pp. 289–306. Springer (2018). https://doi.org/10.1007/978-3-319-96145-3_16
7. Finkbeiner, B., Tentrup, L.: Fast DQBF refutation. In: Proceedings of SAT. LNCS, vol. 8561, pp. 243–251. Springer (2014). https://doi.org/10.1007/978-3-319-09284-3_19
8. Finkbeiner, B., Tentrup, L.: Detecting unrealizability of distributed fault-tolerant systems. Logical Methods in Computer Science **11**(3) (2015). https://doi.org/10.2168/LMCS-11(3:12)2015
9. Fröhlich, A., Kovásznai, G., Biere, A.: A DPLL algorithm for solving DQBF. In: Proceedings of POS@SAT (2012)
10. Fröhlich, A., Kovásznai, G., Biere, A., Veith, H.: iDQ: Instantiation-based DQBF solving. In: Proceedings of SAT. EPiC Series in Computing, vol. 27, pp. 103–116. EasyChair (2014)
11. Gitina, K., Reimer, S., Sauer, M., Wimmer, R., Scholl, C., Becker, B.: Equivalence checking of partial designs using dependency quantified Boolean formulae. In: Proceedings of ICCD. pp. 396–403. IEEE Computer Society (2013). https://doi.org/10.1109/ICCD.2013.6657071
12. Gitina, K., Wimmer, R., Reimer, S., Sauer, M., Scholl, C., Becker, B.: Solving DQBF through quantifier elimination. In: Proceedings of DATE. pp. 1617–1622. ACM (2015)
13. Hecking-Harbusch, J., Tentrup, L.: Solving QBF by abstraction. In: Proceedings of GandALF. EPTCS, vol. 277, pp. 88–102 (2018). https://doi.org/10.4204/EPTCS.277.7
14. Jacobs, S., Basset, N., Bloem, R., Brenguier, R., Colange, M., Faymonville, P., Finkbeiner, B., Khalimov, A., Klein, F., Michaud, T., Pérez, G.A., Raskin, J., Sankur, O., Tentrup, L.: The 4th reactive synthesis competition (SYNT-COMP 2017): Benchmarks, participants & results. In: Proceedings of SYNT@CAV. EPTCS, vol. 260, pp. 116–143 (2017). https://doi.org/10.4204/EPTCS.260.10
15. Jacobs, S., Bloem, R., Brenguier, R., Khalimov, A., Klein, F., Könighofer, R., Kreber, J., Legg, A., Narodytska, N., Pérez, G.A., Raskin, J., Ryzhyk, L., Sankur, O., Seidl, M., Tentrup, L., Walker, A.: The 3rd reactive synthesis competition (SYNT-COMP 2016): Benchmarks, participants & results. In: Proceedings of SYNT@CAV. EPTCS, vol. 229, pp. 149–177 (2016). https://doi.org/10.4204/EPTCS.229.12

16. Janota, M., Klieber, W., Marques-Silva, J., Clarke, E.M.: Solving QBF with counterexample guided refinement. Artif. Intell. **234**, 1–25 (2016). https://doi.org/10.1016/j.artint.2016.01.004
17. Janota, M., Marques-Silva, J.: Solving QBF by clause selection. In: Proceedings of IJCAI. pp. 325–331. AAAI Press (2015)
18. Kleine Büning, H., Karpinski, M., Flögel, A.: Resolution for quantified Boolean formulas. Inf. Comput. **117**(1), 12–18 (1995). https://doi.org/10.1006/inco.1995.1025
19. Korovin, K.: iProver - an instantiation-based theorem prover for first-order logic (system description). In: Proceedings of IJCAR. LNCS, vol. 5195, pp. 292–298. Springer (2008). https://doi.org/10.1007/978-3-540-71070-7_24
20. Peterson, G., Reif, J., Azhar, S.: Lower bounds for multiplayer non-cooperative games of incomplete information. Computers and Mathematics with Applications **41**, 957–992 (2001)
21. Pulina, L., Seidl, M.: The 2016 and 2017 QBF solvers evaluations (QBFEVAL'16 and QBFEVAL'17). Artif. Intell. **274**, 224–248 (2019). https://doi.org/10.1016/j.artint.2019.04.002
22. Rabe, M.N.: A resolution-style proof system for DQBF. In: Proceedings of SAT. LNCS, vol. 10491, pp. 314–325. Springer (2017). https://doi.org/10.1007/978-3-319-66263-3_20
23. Rabe, M.N., Tentrup, L.: CAQE: A certifying QBF solver. In: Proceedings of FMCAD. pp. 136–143. IEEE (2015)
24. Tentrup, L.: Non-prenex QBF solving using abstraction. In: Proceedings of SAT. LNCS, vol. 9710, pp. 393–401. Springer (2016). https://doi.org/10.1007/978-3-319-40970-2_24
25. Tentrup, L.: On expansion and resolution in CEGAR based QBF solving. In: Proceedings of CAV. LNCS, vol. 10427, pp. 475–494. Springer (2017). https://doi.org/10.1007/978-3-319-63390-9_25
26. Tentrup, L., Rabe, M.N.: Clausal abstraction for DQBF (full version). CoRR **abs/1808.08759** (2019), `http://arxiv.org/abs/1808.08759`
27. Wimmer, R., Gitina, K., Nist, J., Scholl, C., Becker, B.: Preprocessing for DQBF. In: Proceedings of SAT. LNCS, vol. 9340, pp. 173–190. Springer (2015). https://doi.org/10.1007/978-3-319-24318-4_13
28. Wimmer, R., Reimer, S., Marin, P., Becker, B.: HQSpre - an effective preprocessor for QBF and DQBF. In: Proceedings of TACAS. LNCS, vol. 10205, pp. 373–390 (2017). https://doi.org/10.1007/978-3-662-54577-5_21