# Incremental Determinization

Markus N. Rabe and Sanjit A. Seshia

University of California, Berkeley
{rabe,sseshia}@berkeley.edu

**Abstract.** We present a novel approach to solve quantified boolean formulas with one quantifier alternation (2QBF). The algorithm incrementally adds new constraints to the formula until the constraints describe a unique Skolem function - or until the absence of a Skolem function is detected. Backtracking is required if the absence of Skolem functions depends on the newly introduced constraints. We present the algorithm in analogy to search algorithms for SAT and explain how propagation, decisions, and conflicts are lifted from *values* to *Skolem functions.* The algorithm improves over the state of the art in terms of the number of solved instances, solving time, and the size of the certificates.

## 1 Introduction

Solvers for quantified boolean formulas (QBFs) have been considered as an algorithmic backend in a variety of application areas, such as planning in uncertain environments [3,32,38], chess [2,3,44], program verification [5,14], model checking of Markov chains [42], circuit analysis [17,18,35], and synthesis [12,16,46]. However, the performance of the currently available solvers can be unsatisfactory. For example, competitive solvers such as DepQBF [34], RAReQS [26], and Qesto [27] cannot solve the quantified boolean formula $\forall X.\, \exists Y.\, X = Y$ in a reasonable timeframe, where $X$ and $Y$ are 32-bit words and $=$ states their bitwise equivalence. Even though preprocessors like Bloqqer [11] help to solve this formula, the example suggests that there is a fundamental problem with the solving principle of state-of-the-art QBF solvers.

The formula describes a trivial problem. We can see that for every assignment to $X$ there is exactly one assignment to $Y$ that satisfies the constraint. That is, the formula describes the Skolem function that is the solution to the problem. This reasoning, however, requires us to detect functional dependencies in formulas that are typically given in conjunctive normal form.

In this paper we present an algorithm to determine the truth of formulas with one quantifier alternation (2QBF) that detects existing functional dependencies among variables and incrementally builds new Skolem functions whenever the problem does not imply a unique Skolem function. We employ the view that the propositional part $\varphi$ of a 2QBF $\forall x_1, \dots, x_n \in \mathbb{B}.\, \exists y_1, \dots, y_m \in \mathbb{B}.\, \varphi$ is a binary *relation* $R_\varphi$ over assignments $\mathbf{x}$ and $\mathbf{y}$ to the variables $x_1, \dots, x_n$ and $y_1, \dots, y_m$: $R_\varphi = \{(\mathbf{x}, \mathbf{y}) \mid \varphi(\mathbf{x}, \mathbf{y})\}$. We call $R_\varphi$ the *Skolem relation.* The solution to a true 2QBF is a *Skolem function* $f$ that assigns values to the existentially quantified variables depending on the universally quantified variables such that the constraints are satisfied for all pairs of assignments $(\mathbf{x}, f(\mathbf{x}))$. Also a Skolem function can be seen as a relation over assignments and it is a subset of the Skolem relation $R_\varphi$. The difference between the Skolem relation $R_\varphi$ and a Skolem function $f$ is that $R_\varphi$ may still provide multiple possible assignments $\mathbf{y}$ for some assignment $\mathbf{x}$, while $f$ has to provide exactly one $\mathbf{y}$ for every $\mathbf{x}$. The presented algorithm adds constraints to $\varphi$ to eliminate the remaining nondeterminism - we *determinize* the Skolem relation to obtain a Skolem function.

The algorithm is a generalization of the DPLL algorithm [15] with conflict-driven clause learning (CDCL) [45]. We lift the concepts of propagation, decisions, and conflicts from *values* for variables to *Skolem functions* for variables. We thereby break the search for Skolem functions down to single variables, which allows us to determinize the relation incrementally, giving rise to the name of the algorithm - *incremental determinization*.

After presenting an overview of the algorithm in Section 3, we present a propagation procedure in Section 4, which identifies variables that already have unique Skolem functions and whether there is a conflicted variable. In Section 5 we discuss how to introduce additional constraints to fix a Skolem function for a variable in case propagation cannot derive a unique Skolem function. Section 6 covers how to compute a conflict clause after a conflicted variable is detected. Termination, correctness, and the generation of certificates is covered in Section 7. In Section 8 we describe the implementation and give an experimental evaluation of the approach. We sketch out relations to other algorithms and preprocessing techniques for QBF in Section 9 and conclude with Section 10.

## 2 Quantified Boolean Formulas

We assume that the reader is familiar with the natural semantics of propositional boolean formulas and summarize the basic notation for quantified boolean formulas in the following. Quantified boolean formulas over a finite set of variables $x \in X$ with domain $\mathbb{B} = \{0, 1\}$ are generated by the following grammar:

$$\varphi \coloneqq 0 \mid 1 \mid x \mid \neg\varphi \mid (\varphi) \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \exists x.\, \varphi \mid \forall x.\, \varphi \ ,$$

We abbreviate multiple quantifications $Qx_1.Qx_2.\ldots Qx_n.\varphi$ to the quantification over a set of variables $QX.\varphi$, where $x_i \in X$ and $Q \in \{\forall, \exists\}$.

An *assignment* $\mathbf{x}$ to a set of variables $X$ is a function $\mathbf{x} : X \to \mathbb{B}$ that maps each variable $x \in X$ to either 1 or 0. Given a propositional formula $\varphi$ over variables $X$ and an assignment $\mathbf{x}'$ for $X' \subseteq X$, we define $\varphi(\mathbf{x}')$ to be the formula obtained by replacing the variables $X'$ by their truth value in $\mathbf{x}'$. By $\varphi(\mathbf{x}', \mathbf{x}'')$ we denote the replacement by multiple assignments for disjoint sets $X', X'' \subseteq X$.

The dependency set of an existentially quantified variable $y$, denoted by $dep(y)$, is the set of universally quantified variables $x$ such that $\exists y.\, \varphi$ is a subformula of $\forall x.\varphi'$. A *Skolem function* $f_y$ maps assignments to $dep(y)$ to assignments to $y$. We define the truth of a QBF $\varphi$ as the existence of Skolem functions $f_Y = \{f_{y_1}, \ldots, f_{y_n}\}$ for the existentially quantified variables $Y = \{y_1, \ldots, y_n\}$, such that $\varphi(\mathbf{x}, f_Y(\mathbf{x}))$ holds for every $\mathbf{x}$, where $f_Y(\mathbf{x})$ is the assignment to $Y$ that the Skolem functions $f_Y$ provide for $\mathbf{x}$.

A quantifier $Q\,x.\,\varphi$ for $Q \in \{\exists, \forall\}$ *binds* the variable $x$ in its subformula $\varphi$. A *closed* QBF is a formula in which all variables are bound. A formula is in prenex normal form, if the formula is closed and starts with a sequence of quantifiers followed by a propositional subformula. A formula $\varphi$ is in the $k$QBF fragment for $k \in \mathbb{N}^+$ if it is closed, in prenex normal form, and has exactly $k-1$ alternations between $\exists$ and $\forall$ quantifiers.

A *literal* $l$ is either a variable $x \in X$, or its negation $\neg x$. Given a set of literals $\{l_1, \ldots, l_n\}$, their disjunction $(l_1 \vee \ldots \vee l_n)$ is called a *clause* and their conjunction $(l_1 \wedge \ldots \wedge l_n)$ is called a *cube*. A propositional formula is in conjunctive normal form (CNF), if it is a conjunction of clauses. A prenex QBF is in prenex conjunctive normal form (PCNF) if its propositional subformula is in CNF. W.l.o.g. we assume for all PCNF formulas that none of the clauses contains two opposite literals, which would trivially satisfy the clause, and that all clauses contain at least one literal from an existentially quantified variable. To simplify the notation, we treat the propositional formulas $\psi$ as sets of clauses $\psi = \{C_1, \ldots, C_n\}$, clauses $C$ as sets of literals $C = \{l_1, \ldots, l_m\}$, and use set operations like intersection and union for their manipulation. Every QBF $\varphi$ can be transformed into an equivalent PCNF with size $O(|\varphi|)$ [47].

We assume that the reader is familiar with unit propagation and define $\mathit{UP}(\varphi)$ as the partial assignment to the variables in a propositional $\varphi$ resulting from applying the unit propagation rule until a fixpoint is reached. We define $\mathit{UP}(\varphi) = \bot$ if unit propagation results in a conflicting assignment for a variable.

## 3 Algorithm

Let $\forall X. \exists Y. \varphi$ be a 2QBF in PCNF, where $\varphi$ is the propositional part. The algorithm IncrementalDeterminization determines whether the formula is true. The key principle of the algorithm is to maintain a set of variables $D \subseteq Y$ for which the set of clauses $\mathcal{D} = \{C \in \varphi \mid C \subseteq D \cup X\}$ that only have variables in $D$ and $X$ defines a Skolem function for each variable in $D$: We say that $\varphi$ is $D$-*consistent* if for each assignment $\mathbf{x}$ to $X$, $\mathit{UP}(\mathcal{D}(\mathbf{x}))$ is not $\bot$ and assigns a value to all variables in $D$. (In particular, $\forall X \exists! D.\ \mathcal{D}$ .) It is clear that a $Y$-consistent 2QBF is true and for each true 2QBF $\forall X. \exists Y. \varphi$ there exists a set of clauses $\psi$, such that $\forall X. \exists Y. \varphi \wedge \psi$ is $Y$-consistent.

Given a $D$-consistent formula $\forall X. \exists Y. \varphi$, we say a variable $v \in Y$ has a *unique Skolem function*, if $\forall X. \exists Y. \varphi$ is also $(D \cup \{v\})$-consistent. For determining $(D \cup \{v\})$-consistency we have to extend the clauses $\mathcal{D}$ by the clauses $\mathcal{U}_v$ in which $v$ is the only variable not in $D$ and not in $X$. Clauses in $\mathcal{U}_v$ can be read as implications where the consequence is a literal of $v$, because we know that all other variables are already determined for all assignments $\mathbf{x}$. We say that a clause $C \in \mathcal{U}_v$ has the *unique consequence* $v$.

The algorithm checks for unique Skolem functions in two steps which require the following definitions: Variable $v$ is *deterministic*, if $\mathit{UP}(\mathcal{D}(\mathbf{x}) \wedge \mathcal{U}_v(\mathbf{x}))$ is $\bot$

or gives a unique assignment to $v$ for all assignments $\mathbf{x}$ to $X$, and $v$ is *conflicted*, if $UP(\mathcal{D}(\mathbf{x}) \wedge \mathcal{U}_v(\mathbf{x})) = \bot$ for some assignment $\mathbf{x}$ to $X$. Deterministic variables that are not conflicted have a unique Skolem function.

```
1: procedure IncrementalDeterminization($\forall X.\exists Y.\varphi$)
2:     $dlvl \leftarrow 0$;  $D \leftarrow \emptyset$
3:     while true do
4:         $D, \varphi, conflict, \mathbf{x} \leftarrow$ Propagate($\forall X.\exists Y.\varphi, D, dlvl$)
5:         if $conflict$ then
6:             $c \leftarrow$ AnalyzeConflict($\forall X.\exists Y.\ \varphi, \mathbf{x}, D, dlvl$)
7:             if $c$ only contains variables in $X$ then
8:                 return false
9:             $dlvl \leftarrow$ (maximal decision level in $c$) $- 1$
10:            $\varphi, D \leftarrow$ Backtrack($\varphi, D, dlvl$)
11:            $\varphi \leftarrow \varphi \wedge c$
12:        else
13:            if $D = Y$ then
14:                return true
15:            $v \leftarrow$ PickVar($Y \setminus D$)
16:            $dlvl \leftarrow dlvl + 1$
17:            $\varphi \leftarrow \varphi \wedge$ Decision($v, \varphi, D$)
```

The elements of the algorithm are as follows: The procedure Propagate extends $D$ with variables with unique Skolem functions until the procedure returns an updated set $D$, or until a conflicted variable is detected upon which propagation reports an assignment to $X$ for which the conflict occurs (see Section 4). In case of a conflict, AnalyzeConflict computes a conflict clause (Section 6).

Variables that are detected to have a unique Skolem function get labeled with the current decision level ($dlvl$) during propagation, which is used during backtracking (lines 10 to 12). The procedure Backtrack($\varphi$, $dlvl$) resets the set $D$ and the formula $\varphi$ to a certain decision level, but keeps the learnt clauses. In case no conflict is detected during propagation, the procedures PickVar and Decision fix a Skolem function for an additional variable (see Section 5).

## 4 Propagation and Conflicts

During *propagation* search algorithms for SAT consider which assignments to the yet unassigned variables are entailed by the current partial assignment. In this section we generalize propagation of values to a notion of propagation of Skolem functions. To develop some intuition on the determinicity check let us consider the following example:

$$\forall x_1.\forall x_2.\ \exists y_1.\exists y_2.\ \underbrace{(x_1 \vee \neg y_1) \wedge (x_2 \vee \neg y_1) \wedge (\neg x_1 \vee \neg x_2 \vee y_1)}_{f_{y_1}:\ (x_1,x_2)\mapsto x_1 \wedge x_2} \tag{1}$$

$$\wedge \underbrace{(\neg x_1 \vee \neg y_2) \wedge (\neg y_1 \vee \neg y_2) \wedge (x_1 \vee y_1 \vee y_2)}_{f_{y_2}:\ (x_1,y_1)\mapsto \neg(x_1 \vee y_1)} \tag{2}$$

The first three clauses (clause group (1)) of the formula can easily be identified as a definition of a Skolem function for $y_1$: For each assignment to $x_1$ and $x_2$, one of the first three clauses entails a unique value for $y_1$. It helps to consider each of these clauses as an implication, that is $(\neg x_1 \rightarrow \neg y_1) \wedge (\neg x_2 \rightarrow \neg y_1) \wedge (x_1 \wedge x_2 \rightarrow y_1)$. Variable $y_1$ thus satisfies the determinicity condition for the empty set $D$. It is also easy to see that the antecedents of the implications described by clause group 1 do not overlap; variable $v$ is not conflicted. We can conclude that there is no solution to the formula above in which $y_1$ has a Skolem function different from $f_{y_1} : (x_1, x_2) \mapsto x_1 \wedge x_2$.

After we identified that $y_1$ has a unique Skolem function, we see that also the Skolem function for variable $y_2$ is unique. The second group of clauses (clause group (2)) allows no Skolem function other than $f_{y_2} : (x_1, y_1) \mapsto \neg(x_1 \vee y_1)$. The use of an existentially quantified variable in the definition of this Skolem function is a short form for the Skolem function $f_{y_2} : (x_1, x_2) \mapsto \neg(x_1 \vee f_{y_1}(x_1, x_2))$.[1] Note that variable $y_2$ does not have a unique Skolem function relative to an empty set $D$. Identifying Skolem functions for some variables can thus help to identify variables for further variables.

### 4.1 Checking for Determinicity

We consider $D$-consistent 2QBF in PCNF $\forall X.\exists Y.\varphi$, where $\varphi$ is the propositional part and $D \subset Y$. For determining determinicity of a variable $v \in Y \setminus D$, only the clauses $\mathcal{D}$ and $\mathcal{U}_v$ play a role. As explained in Section 3 we can see each clause $C$ in $\mathcal{U}_v$ as an implication $\neg(C \setminus \{v, \neg v\}) \implies v$ from variables in $X$ and variables with a fixed Skolem function to a literal of $v$. If, and only if, for every pair of assignments $(\mathbf{x}, \mathbf{d})$ (to variables $X$ and $D$) satisfying $\mathcal{D}$ one of the antecedents described by $\mathcal{U}_v$ applies, variable $v$ is deterministic: $\mathcal{D} \implies \bigvee_{C \in \mathcal{U}_v} \neg(C \setminus \{v, \neg v\})$. To enable the use of SAT solvers we avoid the validity in the formulation and negate the formula.

**Lemma 1.** *Variable $v$ is deterministic w.r.t. $D$ iff the following is unsatisfiable:*

$$\mathcal{D} \;\wedge \bigwedge_{C \in \, \mathcal{U}_v} C \setminus \{v, \neg v\}$$

### 4.2 Local Under-Approximation of Determinicity

Checking determinicity with the formula above can be costly, as the check involves the potentially large set of clauses $\mathcal{D}$. We thus suggest to drop $\mathcal{D}$ and obtain the under-approximation: $\bigwedge_{C \in \, U_v} C \setminus \{v, \neg v\}$ . That is, the local under-approximation does not take into account that certain assignments to $D$ may violate the definitions of the Skolem functions. We call a variable that satisfies the local determinicity check *locally deterministic.*

[1] The algorithm only constructs Skolem functions that depend on universally quantified variables. The use of existentially quantified variables is just an abbreviation, so there is no risk of circular dependencies.

Let us revisit the example in the beginning of Section 4. Assuming that none of the variables have been identified yet to have a unique Skolem function ($D = \emptyset$), we check local determinicity for variable $y_1$ with the following propositional formula: $x_1 \wedge x_2 \wedge (\neg x_1 \vee \neg x_2)$. As the formula is unsatisfiable, $y_1$ is deterministic. If we checked $y_2$ first, there would be only a single clause with unique consequence $y_2$ and thus the local determinicity check consists of the single clause $\neg x_1$, which is trivially satisfiable. Only after identifying a Skolem function for $y_1$, i.e. with $y_1 \in D$, also the other two clauses in which $y_2$ occurs have a unique consequence. In this case we formulate the query $\neg x_1 \wedge \neg y_1 \wedge (x_1 \vee y_1)$, determine its unsatisfiability, and conclude that $y_2$ satisfies condition (1) as well.

### 4.3 Pure Literals

If an existentially quantified variable occurs in only one polarity in a formula in PCNF we call it a *pure literal* and we can assign it this polarity while preserving the truth of the formula. When we additionally consider a partial assignment, we can easily generalize this to the following: when all literals of one polarity are in clauses that are satisfied by literals of other variables, we can assign the variable the opposite literal. We encode this condition as a constraint:

**Lemma 2.** *Given a PCNF $\forall X.\exists Y.\varphi$ and a literals $l$ of $v \in Y$, we have:*

$$\forall X.\exists Y.\varphi \iff \forall X.\exists Y.\varphi \wedge \Big( \big( \bigwedge_{C\in\varphi \text{ with } l\in C} C \setminus \{l\} \big) \implies \bar{l} \Big)$$

We call $\left(\bigwedge_{C\in\varphi \text{ with } l\in C} C \setminus \{l\}\right) \implies \bar{l}$ the *pure literal constraint*. We could add the constraint for all variables, but this would increase the formula size significantly. Instead we add the constraint only if during propagation $v$ is not locally deterministic and a literal $l$ of $v$ only occurs in clauses in $\mathcal{U}_v$. Adding the pure literal constraint then guarantees that $v$ is deterministic.

Consider a variation of the previous example, where we only flip the negation of the second occurrence of $y_2$:

$$\forall x_1.\forall x_2.\ \exists y_1.\exists y_2.\ \underbrace{(x_1 \vee \neg y_1) \wedge (x_2 \vee \neg y_1) \wedge (\neg x_1 \vee \neg x_2 \vee y_1)}_{f_{y_1}\colon\ (x_1,x_2)\mapsto x_1\wedge x_2} \tag{3}$$

$$\wedge \underbrace{(\neg x_1 \vee \neg y_2)}_{f_{y_2}\colon\ x_1\mapsto\neg x_1} \wedge (\neg y_1 \vee y_2) \wedge (x_1 \vee y_1 \vee y_2) \tag{4}$$

Even when we test $y_2$ for determinicity before we establish that $y_1$ is deterministic, we can now fix a Skolem function for $y_2$: The only negative occurrence of variable $y_2$ is in the clause $\neg x_1 \vee \neg y_2$, which happens to have $\neg y_2$ as a unique consequence. We can thus fix $y_2$ to be positive in all remaining cases and set $f_{y_2} :\ x_1 \mapsto \neg x_1$ by adding the clause $x_1 \vee y_2$ to the formula.

## 4.4 Checking for Conflicts

In algorithms for propositional SAT, we call variables conflicted, if unit propagation resulted in conflicting assignments to the variable. In the incremental determinization algorithm a variable $v$ is conflicted in a $D$-consistent 2QBF, if there is an assignment to the universally quantified variables that propagates conflicting assignments to $v$. We consider the example from Subsection 4.3 to develop some intuition on conflicted variables. Assume we first identified variable $y_2$ to have the unique Skolem function $f_{y_2} : \ x_1 \mapsto \neg x_1$. Then all clauses in clause group 3 and the latter two clauses of clause group 4 have a literal of $y_1$ as a unique consequence. The determinicity check determines that the formula $x_1 \wedge x_2 \wedge (\neg x_1 \vee \neg x_2) \wedge \neg y_2 \wedge (x_1 \vee y_2)$ is unsatisfiable and concludes that $y_1$ is deterministic. To see that the variable is conflicted, i.e. there is no Skolem function satisfying all constraints, consider the assignment $x_1 \wedge x_2$, which sets $y_2$ to false according to $f_{y_2}$. In this case we cannot give variable $y_1$ a value that satisfies all constraints: The last clause of clause group 3 requires $y_1$ to be set to true while the second clause of clause group 4 requires $y_1$ to be set false. We found a conflict!

As for the determinicity check, we can easily construct a formula for the *global conflict check* that represents the assignments to $X$ that prove a variable $v \in Y \setminus D$ to be conflicted. Let us consider a $D$-consistent 2QBF in PCNF $\forall X. \exists Y. \varphi$, where $\varphi$ is the propositional part and $D \subset Y$. The clauses $\mathcal{D}$ represent the known Skolem functions and are guaranteed to provide unique values to $D$ for every assignment $\mathbf{x}$ to $X$. In particular, $UP(\mathbf{x})$ cannot result in $\bot$ and so it suffices to check for conflicting assignments to variable $v$. Variable $v$ can only be propagated if one of the antecedents of the clauses $\mathcal{U}_v$ with unique consequence $v$ is true. We thus know that variable $v$ is conflicted if, and only if, there is a pair of assignments $(\mathbf{x}, UP(\mathbf{x}))$ that satisfies the antecedents of two clauses $C, C' \in \mathcal{U}_v$ with $v \in C$ and $\neg v \in C'$.

**Lemma 3.** *Variable $v$ is conflicted if, and only if, the following is satisfiable:*

$$\mathcal{D} \wedge \left( \bigvee_{C \in \mathcal{U}_v \ \textit{with} \ v \in C} \neg (C \setminus \{v\}) \right) \wedge \left( \bigvee_{C \in \mathcal{U}_v \ \textit{with} \ \neg v \in C} \neg (C \setminus \{\neg v\}) \right)$$

## 4.5 Local Over-Approximation for Conflict Detection

The global conflict check is a relatively expensive step, as it involves the potentially large set of clauses $\mathcal{D}$. Similar to the local determinicity check, we drop $\mathcal{D}$ from the conflict check to first check for local conflicts. If the local conflict check returns an assignment to $X$ and $D$, we cannot be sure that the assignment satisfies $\mathcal{D}$, so we then resort to the global conflict check.

## 4.6 The Propagation Procedure

The procedure PROPAGATE extends a given set $D$ of variables that have a unique Skolem function and it checks whether there is a conflicted variable. The returned set $D$ may be an under-approximation of the set of variables with unique Skolem functions, just as propagation for SAT computes an under-approximation of the set of variables having a unique value. This may lead to unnecessary decisions, but avoids the costly global determinicity check. Also, the procedure does not check all variables for conflicts. Instead it only makes sure that deterministic variables are not conflicted, so no conflicted variable gets added to the set $D$. In this way all variables will still be checked for conflicts eventually (unless the algorithm terminates with false).

Given a set $D$ of variables, a $D$-consistent 2QBF, and a decision level, Propagate returns a 4-tuple indicating the updated set of variables $D$, whether there is a conflict, the formula (which may be modified by pure literal detection), and an assignment $\mathbf{x}$ to the universally quantified variables:

```
1:  procedure Propagate($\forall X.\exists Y.\ \varphi$, $D$, $dlvl$)
2:      $U \leftarrow$ variables occurring as unique consequence in $\varphi$
3:      while $U \cap (Y \setminus D) \neq \emptyset$ do
4:          $v \leftarrow$ pick a variable in $U \cap (Y \setminus D)$
5:          $U \leftarrow U \setminus v$
6:          if $v$ is locally deterministic then
7:              if local conflict for $v$ then
8:                  if global conflict for $v$ for assignment $\mathbf{x}$ then
9:                      return (D, true, $\varphi$, $\mathbf{x}$)
10:             $v.dlvl \leftarrow dlvl$
11:             $D \leftarrow D \cup \{v\}$
12:             check for new clauses with unique consequences; update $U$
13:         else
14:             if $v$ occurs only in $U_v$ or $\neg v$ occurs only in $U_v$ then
15:                 $\varphi \leftarrow \varphi \wedge$ pure literal constraint
16:                 $U \leftarrow U \cup \{v\}$
17:     return ($D$, false, $\varphi$, N/A)
```

With the set $U$ we remember which variables we still have to check for determinicity. Whenever a variable is detected to have a unique Skolem function, we check for clauses that now have a unique consequence and update $U$ (line 12). It is possible (and desirable) to start with a smaller set $U$ than shown above: only variables $v$ for which we added a new clause with unique consequence $v$ since the last propagation phase can possibly become deterministic. For the sake of simplicity we omitted the additional bookkeeping in this exposition.

## 5 Decisions

Decisions are made when the propagation procedure comes to a stop and no conflict was detected. The procedure PickVar picks a variable $v \in Y \setminus D$, which we call the *decision variable*. The procedure Decision then adds clauses, the *decision clauses*, that make variable $v$ locally deterministic. Note the decision variable may be conflicted, though not yet detected as such, at the time of the decision, as the propagation procedure does not guarantee that all variables are conflict free. In the next propagation phase, after the decision variable is detected to be deterministic, it may thus be detected to be conflicted.

In this section we propose a simple way to take decisions that avoids introducing additional conflicts—between decision clauses and clauses in $\mathcal{U}_v$—for the decision variable. We simply fix the Skolem function that assigns 1 to $v$ whenever the clauses $\mathcal{U}_v$ do not require otherwise. That is, we consider the clauses with unique consequence that may require $v$ to be set to 0, i.e. $C \in \mathcal{U}_v$ with $\neg v \in C$, and define the result of Decision as the constraint that sets $v$ to 1 when all their antecedents are false:

$$\Big( \bigwedge_{C \in \mathcal{U}_v \, with \neg v \in C} C \setminus \{\neg v\} \Big) \implies v$$

We again consider the example of Subsection 4.3 with an empty set $D$. In case we were to take a decision over variable $y_2$ instead of considering the pure literal rule, we would fix the cases described by the only clause in $\mathcal{U}_v$, which is $\neg x_1 \vee \neg y_2$. Then we fix $y_2$ to be true in all remaining cases, i.e. by adding the clause $x_1 \vee y_2$.

Given a $D$-consistent 2QBF $\forall X.\exists Y.\varphi$ it is clear that a variable $v \in Y \setminus D$ is deterministic in $\forall X.\exists Y.\varphi \wedge \textsc{Decision}(v, \varphi, D)$. It is also easy to see that the procedure does not introduce additional conflicts for $v$. Also, for all assignments $\mathbf{x}$ to $X$ and $\mathbf{d}$ to $D$ we have:

$$UP\big(\mathcal{D} \wedge \mathcal{U}_v \wedge \textsc{Decision}(v, \varphi, D)(\mathbf{x}, \mathbf{d})\big) = \bot \implies UP\big(\mathcal{D} \wedge \mathcal{U}_v(\mathbf{x}, \mathbf{d})\big) = \bot$$

This property guarantees that the conflict analysis, which we cover in Section 6, always results in a *new* clause and thereby provides us with an argument for the termination of the algorithm. The procedure Decision also marks the added clauses as *decision clauses*. During conflict analysis and during backtracking we have to distinguish decision clauses from learnt clauses.

## 6 Conflict Analysis

*Conflict analysis* for incremental determinization stays remarkably similar to CDCL. Once a (global) conflict is detected, we compute a conflict clause along an implication graph. If the conflict clause contains only universally quantified variables, we proved the formula to be false. Otherwise, we have to backtrack to the largest decision level that contributed to the conflict, add the conflict clause to the formula, and continue with propagation.

Let us consider a $D$-consistent 2QBF $\forall X.\exists Y.\ \varphi$, and let $\delta \subseteq \varphi$ be the set of decision clauses for the decision variables $E \subseteq Y$. Assume we detected a conflict for a variable $v \in Y \setminus D$ for which the global conflict check returned the assignment $\mathbf{x}$ to $X$. The procedure AnalyzeConflict first computes $UP(\mathcal{D}(\mathbf{x}))$ to obtain an assignment $\mathbf{e}$ for the variables $D \cap E$. Then the algorithm computes a conflict clause as for CDCL [45] along the implication graph of:

$$UP\Big(\big((\mathcal{D} \wedge \mathcal{U}_v) \setminus \delta\big)(\mathbf{x}, \mathbf{e})\Big)$$

which is guaranteed to return $\bot$. That is, we omit the decision clauses and instead treat decision variables as a decisions in the sense of CDCL with the values obtained by $UP(\mathcal{D}(\mathbf{x}))$.

**Lemma 4.** *Let $\forall X.\exists Y.\ \varphi$ be a D-consistent 2QBF and let $\delta \subseteq \varphi$ be the set of decision clauses in $\varphi$. The algorithm* AnalyzeConflict$(\forall X.\exists Y.\ \varphi, D, dlvl)$ *returns a clause $C$ such that $\varphi \setminus \delta \Leftrightarrow (\varphi \setminus \delta) \wedge C$ and $C \notin \varphi \setminus \delta$.*

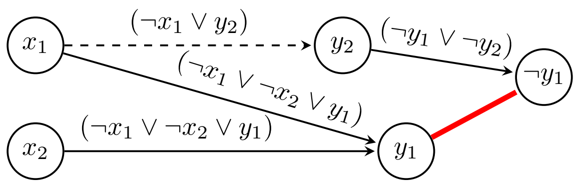
*Proof.* Let $v \in Y \setminus D$ be a conflicted variable, provoked by an assignment $\mathbf{x}$ to $X$. After every decision the first variable that is checked for determinicity and conflictedness, is the decision variable. So we have $E \subseteq (D \cup \{v\})$ and if $v \notin E$ then we even have $E \subseteq D$. In either case $UP(\mathcal{D}(\mathbf{x}))$ returns an assignment to all decision variables that are not conflicted. Since $v$ is conflicting with the decision clauses, it is also conflicting when we replace the decision clauses by the values of the decision variables: $UP\Big(((\mathcal{D} \wedge \mathcal{U}_v) \setminus \delta)(\mathbf{x}, \mathbf{e})\Big) = \bot$.

Any conflict clause $C$ derived by CDCL can be derived by a sequence of resolution steps [40] and we thus know $(\mathcal{D} \wedge \mathcal{U}_v) \setminus \delta \Leftrightarrow ((\mathcal{D} \wedge \mathcal{U}_v) \setminus \delta) \wedge C$ by the soundness of the resolution rule [43]. Since $C$ is over the same variables as $\mathcal{D}$ this extends to $\varphi \Leftrightarrow \varphi \wedge C$. Also, a conflict clause computed as for CDCL is guaranteed to be not contained in the formula it is derived from, so it is not in $(\mathcal{D} \wedge \mathcal{U}_v) \setminus \delta$. Since $C$ only contains variables from $X$ and $D$ and since $\mathcal{D}$ includes all clauses over $X$ and $D$ in $\varphi$, $C$ cannot be in $\varphi \setminus (\mathcal{D} \cup U_v)$ either. $\square$

Consider the following example:

$$\forall x_1.\forall x_2.\ \exists y_1.\exists y_2. \quad (x_2 \vee \neg y_1) \wedge (\neg x_1 \vee \neg x_2 \vee y_1) \tag{5}$$
$$\wedge\, (x_1 \vee \neg y_2) \wedge (\neg y_1 \vee \neg y_2) \wedge (x_1 \vee y_1 \vee y_2) \tag{6}$$

This time, neither $y_1$ nor $y_2$ can be propagated. Let us pick $y_2$ as the decision variable. According to the decisions discussed in Section 5 we add the clause $\neg x_1 \vee y_2$ to complement the only other clause in which $y_2$ occurs as its unique consequence, i.e., $x_1 \vee \neg y_2$. The variable $y_2$ is thus assigned the Skolem function $f_{y_2} : x_1 \mapsto x_1$. Now variable $y_1$ is conflicted: The conflict test could return the assignment for the universal variables represented by the conjunction $x_1 \wedge x_2$, which determines $y_2$ to be true according to $f_{y_2}$. The second clause of clause group 5 and the second clause of clause group 6 then require conflicting assignments to $y_1$. Consider the implication graph:


$x_1$
$(\neg x_1 \vee y_2)$
$y_2$
$(\neg y_1 \vee \neg y_2)$
$\neg y_1$
$(\neg x_1 \vee \neg x_2 \vee y_1)$
$x_2$
$(\neg x_1 \vee \neg x_2 \vee y_1)$
$y_1$

The presence of both nodes $y_1$ and $\neg y_1$ represents a conflict, indicated by the red edge. The labels on the edges indicate the clauses with unique consequence that correspond to the edges. The clause $\neg x_1 \vee \neg y_2$ added in the decision corresponds to the dashed edge from $x_1$ to $y_2$. AnalyzeConflict considers the implication graph for the formula without the dashed edge and instead assumes the value 1 for the decision variable $y_2$ as an additional decision. The only conflict clause that could be computed in this case is $\neg x_1 \vee \neg x_2 \vee \neg y_2$.

## 7 Correctness, Termination, Certificates

**Theorem 1.** *Incremental determinization is correct and terminates.*

*Proof sketch:* We first show that the algorithm maintains $D$-consistency and terminates as there are only finitely many clauses that could be learnt. If the algorithm terminates with true, the computed formula is $Y$-consistent and contains the original formula. This represents a Skolem function satisfying all original constraints. Lemma 4 and Lemma 2 imply soundness for the case that the algorithm terminates with false.

**Certificates.** Once terminated, the correctness of the result can be checked by an independent, simpler algorithm. This is important in practice since highly optimized logic solvers can easily contain programming errors. In some applications the proof object, that is the Skolem function, may also represent an interesting object like an implementation or a strategy.

For the presented method, certification is straightforward: The final set of clauses is $Y$-consistent and we can thus obtain an assignment to the existentially quantified variables $Y$ for every assignment to the universally quantified variables through propagation - the clauses represents the Skolem function. To check the correctness of the Skolem function via existing QBF proof checkers such as QBFCert [36, 41], we translate the function into a circuit in the AIGER format [1]. The circuit reads the values of the universally quantified variables and its outputs provide the values for the existentially quantified variables.

We can exploit the order in which the algorithm added variables to the set $D$. This order provides us a unique direction in which the clauses can propagate. Let $U_v$ be the set of clauses with unique consequence at the time variable $v$ was added last to the set $D$. For every variable $v$ we then define the circuit's output for variable $v$ as

$$\bigvee_{C \in \mathcal{U}_v \ \text{with} v \in C} \neg(C \setminus \{v\}) \ .$$

The order among the existentially quantified variables then guarantees the absence of circular dependencies.

In case the algorithm determines that the given QBF is false, the implementation provides the assignment to the universally quantified variables from the last global conflict check.

| Family | total | CADET - | CADET +b | RAReQS - | RAReQS +b | quantor - | quantor +b | Qesto - | Qesto +b | CAQE - | CAQE +b | DepQBF - | DepQBF +b | GhostQ - |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Terminator | 590 | **583** | 513 | 0 | 382 | 12 | 138 | 1 | 491 | 32 | 288 | 547 | 572 | 480 |
| Hardware Fixpoint | 131 | **110** | 92 | 8 | 67 | 16 | 62 | 8 | 67 | 7 | 78 | 22 | 68 | 8 |
| Ranking Functions | 365 | **365** | **365** | 0 | 363 | 13 | 360 | 0 | 363 | 0 | 363 | 168 | 360 | 6 |
| Reduction Finding | 48 | 0 | 4 | 21 | **38** | 2 | 5 | 22 | 36 | 16 | 26 | 14 | 16 | 2 |
| Circuit Underst. | 78 | 20 | 44 | 2 | **50** | 2 | 6 | 2 | 34 | 2 | 48 | 36 | 17 | 36 |
| Partial Equivalence | 300 | 217 | 201 | 2 | 246 | 80 | 102 | 20 | 235 | 56 | **261** | 86 | 159 | 11 |
| Reactive Synthesis | 153 | **153** | **153** | 133 | **153** | **153** | **153** | 129 | **153** | 106 | **153** | 137 | **153** | 149 |
| Random | 254 | 242 | 243 | 246 | 238 | 90 | 91 | 226 | 220 | 249 | **250** | 231 | 224 | 183 |

**Table 1.** Number of instances solved within 10 minutes and 4 GB memory for various benchmarks. For columns labeled with +b we applied the preprocessor Bloqqer before running the solver. Numbers in bold font indicate the best result for a benchmark.

## 8 Implementation and Experimental Evaluation

We implemented the algorithm in a tool we named the Cal incremental determinizer (CADET)[2], using PicoSAT [10] to solve the propositional problems. The implementation emphasizes simplicity over speed and consists of about 4000 lines of code (not counting the SAT solver) in the programming language C. For global conflict detection, we make use of the incremental solving features of PicoSAT. We maintain an instance of PicoSAT containing the clauses $\mathcal{D}$ that have only literals of universally quantified variables and deterministic existentially quantified variables. Each global conflict check can then be performed by adding only few clauses. Whenever we detect new variables to be deterministic, we push more clauses into the SAT solver, and accordingly pop clauses during backtracking. We implemented two further optimizations:

*Restarts:* After a certain threshold of conflicts, we backtrack all decisions and continue with propagation. The restart threshold is then increased by a constant factor >1. Thus there will eventually be a large enough interval such that the termination argument holds.

*Constant propagation:* When the algorithm starts and whenever we identify variables as deterministic, we check for unit clauses. Unit clauses imply a unique value for a variable, which we propagate among the not yet determined variables.

**Experimental Evaluation.** We performed experiments on several sets of benchmarks of 2QBF instances to compare CADET to state-of-the-art QBF solvers. The experiments were conducted on machines with a quadcore 3.6 GHz Intel Xeon processor, with a 10 minute timeout and 4GB memory limit. Table 1 summarizes the number of instances solved for different benchmark families.

The first group of benchmarks was taken from QBFLIB [19]. These benchmarks consist of interesting software and hardware verification problems (Terminator [5], Hardware Fixpoint [48], Ranking Functions [14]) and demonstrate the strength of incremental determinization compared to existing algorithms.

[2] CADET is available via `https://eecs.berkeley.edu/~rabe/cadet.html`

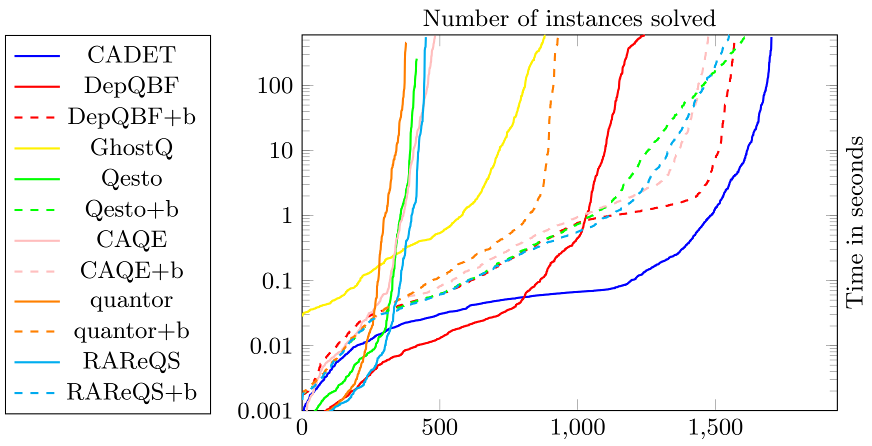


**Fig. 1.** Log-scale cactus plot comparing the performance over all instances.

CADET solved more instances than any other approach while not being dependent on preprocessing.

The second group of benchmarks is from recent papers on QBF applications: Reduction Finding [29] (also used in QBFGallery 2014 [25]), Circuit Understanding [18], Partial Equivalence [17], Reactive Synthesis [12]. We included the second group of benchmarks to also present cases in which incremental determinization is not superior to the existing approaches. The last benchmark, Random, consists of all 2QBFs from the randomly generated instances listed on QBFLIB [19].

Fig. 1 shows that CADET is the strongest solver overall. The log-scale allows us to observe that there is a substantial gap between the solving times of CADET and the other approaches. This is also reflected in the overall solving times, where CADET leads with 144009 seconds before Qesto+b, DepQBF+b, and RAReQS+b taking 224220, 243575, and 259125 seconds.

The scatter plot in Fig. 2 compares the relative runtimes of RAReQS (with Bloqqer) and CADET. The scatter plot suggests that incremental determinization and abstraction refinement perform quite differently and that their relative performance highly depends on the benchmark. The comparison to the other solvers shows similar features.

The certificates computed by CADET are typically much smaller than those by DepQBF. In Fig. 3 we compare the certificate sizes of DepQBF and CADET. For CADET we present the size of the certificates before and after simplification with the ABC model checker. For DepQBF we used the `--simplify` during the generation with `qrpcert`. DepQBF's certificate cannot be simplified further using ABC, as DepQBF encodes some information in the structure of the certificates. The plot reveals an enormous difference in the ability to certify and the quality of the produced certificates.

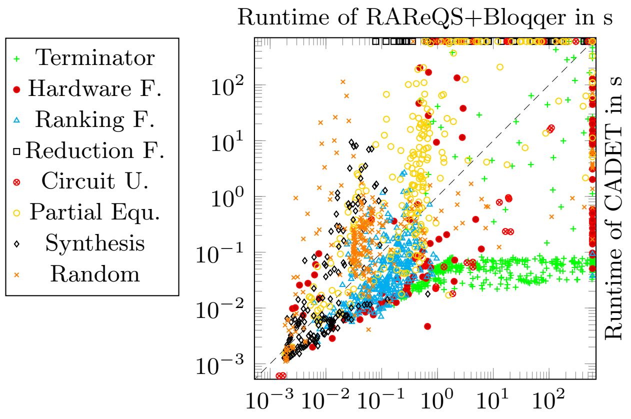


**Fig. 2.** Relative performance of CADET and RAReQS+Bloqqer.

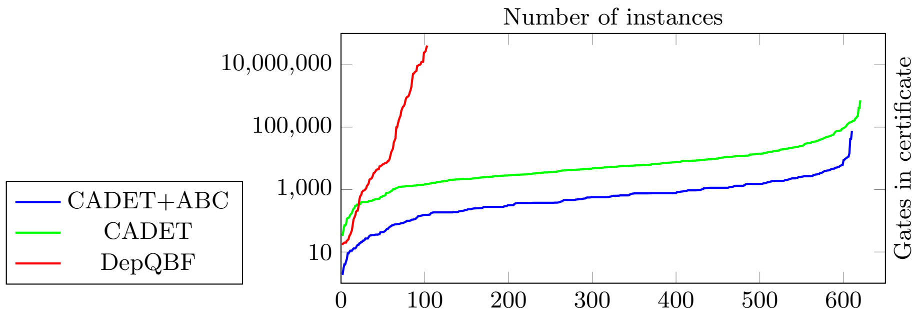


**Fig. 3.** Log-scale cactus plot showing the distribution of sizes of the certificates.

## 9 Related Work

Early approaches to solving QBFs focused on *expanding* the quantifiers using data structures like CNF [9,33], BDDs [4,37] or AIGs [39]. *Skolemization* helps to reduce the expansion effort [7]. *QDPLL* is a generalization of DPLL to QBF where we propagate and decide on values of variables, as for propositional SAT [6, 13, 20, 34]. QDPLL solvers often explore a large portion of the exponentially many assignments to the universally quantified variables as the example from the introduction shows. The incremental determinization algorithm encapsulates the reasoning about the assignments to the universal variables in the global conflict check, which is a propositional formula and can thus be offloaded to SAT solvers.

The *CEGAR* approach for QBF [26–28, 41] was most competitive in recent evaluations [19,25]. The basic idea is to maintain one SAT solver for each quantifier level. In each iteration these algorithms exclude all assignments to universal variables that can be matched with one assignment to the existential variables. Skolem functions that require many different outputs, like the identity example $\forall X. \exists Y.\ X = Y$ from the introduction, thus need an exponential number of iterations. Incremental determinization does not have to explore single assignments to $Y$ in cases, like the identity example, where we can derive parts of the Skolem function directly.

*Non-CNF solvers.* Recently there has been a surge of interest in QBF solvers that are not based on the PCNF representation of formulas. Exploiting the formula structure [21,30,31] or even the word-level structure [48] has been argued to be superior to pure CNF-level reasoning. The experimental results in this paper suggest that this question is not yet settled. For example, the circuit-based solver GhostQ is significantly less effective than CADET on most benchmarks. After the initial circuit-extraction current circuit-based solvers suffer from the same principal problem as solvers using QDPLL or CEGAR.

*Proof systems for QBF.* Resolution based proof calculi, like *resolve and expand* [9] and IR [8], also add clauses to the QBF. The generation of conflict clauses in this work relies on the instantiation of the universally quantified variables, which resembles the proof rules in IR. In contrast, incremental determinization adds constraints that possibly change the truth of the formula. This comes at the cost of backtracking, in case the added constraints (during decisions) were too strong.

*Certification.* Previous certifying QBF solvers produce proof traces from which certificates (Skolem functions) can be reconstructed [36,41]. Also preprocessing techniques such as QBCE [11] can be certified [22,23] and combined with certificates for solvers [24]. In this work, the final set of clauses is the certificate and can easily be translated into a circuit representing the Skolem function.

## 10 Conclusions

Quantified boolean formulas are a natural choice to encode problems in verification, synthesis, and artificial intelligence. We give a completely new algorithm to solve problems in 2QBF that is based on incrementally adding constraints until the Skolem relation collapses to a Skolem function. The algorithm employs a propagation step that directly constructs Skolem functions out of the constraints of the formula. We thereby exploit the formula structure despite working on the level of the CNF. The example from the introduction, $\forall X. \exists Y.\ X = Y$, is thus solved instantly. Working on the level of Skolem functions avoids situations in which other approaches have to explore an exponential number of cases and also helps to certify the results.

Most parts of the algorithm are kept simplistic on purpose. Likely there are stronger and more efficient ways to propagate, take decisions, and to analyze conflicts in this setting. The purpose of this work is to introduce a framework for the direct manipulation of Skolem functions in a search-based algorithm. Still, the experimental evaluation suggests that the algorithm already improves over existing approaches.

**Acknowledgements.** We thank our anonymous reviewers for their comments and Marcell Vazquez-Chanlatte, Leander Tentrup, Ashutosh Trivedi, and Christoph Wintersteiger for fruitful discussions on this work. Adrià Gascón and Leander Tentrup kindly provided 2QBF benchmarks. This work was supported in part by NSF grants CCF-1139138 and CCF-1116993, by SRC contract 2460.001, NSF STARSS grant 1528108, and by gifts from Toyota and Microsoft.

## References

1. AIGER toolset. `http://fmv.jku.at/aiger/`.
2. Rajeev Alur, P Madhusudan, and Wonhong Nam. Symbolic computational techniques for solving games. *Software Tools for Technology Transfer*, 7(2):118–128, 2005.
3. Carlos Ansotegui, Carla P Gomes, and Bart Selman. The Achilles' heel of QBF. In *Proceedings of AAAI*, volume 2, pages 2–1, 2005.
4. Gilles Audemard and Lakhdar Sais. A symbolic search based approach for quantified boolean formulas. In *Proceedings of SAT*, pages 16–30, 2005.
5. Gérard Basler, Daniel Kroening, and Georg Weissenbacher. SAT-based summarization for boolean programs. In *Proceedigns of Model Checking Software*, pages 131–148. Springer, 2007.
6. Sam Bayless and Alan J. Hu. Single-solver algorithms for 2QBF. In *Proceedings of SAT*, pages 487–488, Berlin, Heidelberg, 2012. Springer-Verlag.
7. Marco Benedetti. skizzo: A suite to evaluate and certify QBFs. In *Proceedings of CADE*, pages 369–376, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.
8. Olaf Beyersdorff, Leroy Chew, and Mikoláš Janota. On unification of QBF resolution-based calculi. In *Mathematical Foundations of Computer Science 2014*, pages 81–93. Springer, 2014.
9. Armin Biere. Resolve and expand. In *Proceedings of SAT*, 2004.
10. Armin Biere. PicoSAT essentials. *JSAT*, 4(2-4):75–97, 2008.
11. Armin Biere, Florian Lonsing, and Martina Seidl. Blocked clause elimination for QBF. In *Proceedings of CADE*, pages 101–115, 2011.
12. Roderick Bloem, Uwe Egly, Patrick Klampfl, Robert Könighofer, and Florian Lonsing. SAT-based methods for circuit synthesis. In *Proceedings of FMCAD*, pages 31–34, 2014.
13. Marco Cadoli, Marco Schaerf, Andrea Giovanardi, and Massimo Giovanardi. An algorithm to evaluate quantified boolean formulae and its experimental evaluation. *Journal of Automated Reasoning*, 28(2):101–142, 2002.
14. Byron Cook, Daniel Kroening, Philipp Rümmer, and Christoph M. Wintersteiger. Ranking function synthesis for bit-vector relations. In *Proceedings of TACAS*, pages 236–250, 2010.
15. Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.
16. Peter Faymonville, Bernd Finkbeiner, Markus N Rabe, and Leander Tentrup. 3 encodings of reactive synthesis. In *Proceedings of QUANTIFY*, pages 20–22, 2015.
17. Bernd Finkbeiner and Leander Tentrup. Fast DQBF refutation. In *Proceedings of SAT*, pages 243–251, 2014.
18. Adria Gascón, Pramod Subramanyan, Bruno Dutertre, Anish Tiwari, Dejan Jovanovic, and Sharad Malik. Template-based circuit understanding. In *Proceedings of FMCAD*, pages 83–90. IEEE, 2014.
19. E. Giunchiglia, M. Narizzano, L. Pulina, and A. Tacchella. Quantified Boolean Formulas satisfiability library (QBFLIB), 2005. `www.qbflib.org`.
20. Enrico Giunchiglia, Massimo Narizzano, and Armando Tacchella. QuBE: A system for deciding quantified boolean formulas satisfiability. In *Proceedings of IJCAR*, pages 364–369, 2001.
21. Alexandra Goultiaeva and Fahiem Bacchus. Recovering and utilizing partial duality in QBF. In *Proceedings of SAT*, pages 83–99. Springer, 2013.

22. Marijn Heule, Martina Seidl, and Armin Biere. Efficient extraction of skolem functions from QRAT proofs. In *Proceedings of FMCAD*, pages 107–114, 2014.
23. Marijn Heule, Martina Seidl, and Armin Biere. A unified proof system for QBF preprocessing. In *Proceedings of IJCAR*, volume 8562 of *LNCS*, pages 91–106. Springer, 2014.
24. Mikolás Janota, Radu Grigore, and João Marques-Silva. On QBF proofs and preprocessing. In *Proceedings of LPAR*, pages 473–489, 2013.
25. Mikolas Janota, Charles Jordan, Will Klieber, Florian Lonsing, Martina Seidl, and Allen Van Gelder. The QBFGallery 2014: The QBF competition at the FLoC olympic games. *Journal on Satisfiability, Boolean Modeling and Computation*, 9:187–206, 2016.
26. Mikolás Janota, William Klieber, João Marques-Silva, and Edmund M. Clarke. Solving QBF with counterexample guided refinement. In *Proceedings of SAT*, pages 114–128, 2012.
27. Mikolás Janota and Joao Marques-Silva. Solving QBF by clause selection. In *Proceedings of IJCAI*, pages 325–331. AAAI Press, 2015.
28. Mikolás Janota and João P. Marques Silva. Abstraction-based algorithm for 2QBF. In *Proceedings of SAT*, pages 230–244, 2011.
29. Charles Jordan and Łukasz Kaiser. Experiments with reduction finding. In *Proceedings of SAT*, pages 192–207. Springer, 2013.
30. Charles Jordan, Will Klieber, and Martina Seidl. Non-cnf qbf solving with qcir. In *Proceedings of BNP (Workshop)*. 2016.
31. William Klieber, Samir Sapra, Sicun Gao, and Edmund Clarke. A non-prenex, non-clausal QBF solver with game-state learning. In *SAT*, pages 128–142, Berlin, Heidelberg, 2010. Springer-Verlag.
32. Martin Kronegger, Andreas Pfandler, and Reinhard Pichler. Conformant planning as a benchmark for QBF-solvers. In *Report on the International Workshop on QBF*, pages 1–5, 2013.
33. Florian Lonsing and Armin Biere. Nenofex: Expanding NNF for QBF solving. In *Proceedings of SAT*, pages 196–210, 2008.
34. Florian Lonsing and Armin Biere. DepQBF: A dependency-aware QBF solver. *JSAT*, 7(2-3):71–76, 2010.
35. C. Miller, C. Scholl, and B. Becker. Proving QBF-hardness in bounded model checking for incomplete designs. In *Proceedings of MTV*, pages 23–28, 2013.
36. Aina Niemetz, Mathias Preiner, Florian Lonsing, Martina Seidl, and Armin Biere. Resolution-based certificate extraction for QBF - (tool presentation). In *Proceedings of SAT*, pages 430–435, 2012.
37. Oswaldo Olivo and E. Allen Emerson. A more efficient BDD-based QBF solver. In *Proceedings of CP*, pages 675–690, 2011.
38. Charles Otwell, Anja Remshagen, and Klaus Truemper. An effective QBF solver for planning problems. In *MSV/AMCS*, pages 311–316. Citeseer, 2004.
39. Florian Pigorsch and Christoph Scholl. An AIG-based QBF-solver using SAT for preprocessing. In *Proceedings of DAC*, pages 170–175. IEEE, 2010.
40. Knot Pipatsrisawat and Adnan Darwiche. On the power of clause-learning {SAT} solvers as resolution engines. *Artificial Intelligence*, 175(2):512 – 525, 2011.
41. Markus N Rabe and Leander Tentrup. CAQE: A certifying QBF solver. In *Proceedings of FMCAD*, pages 136–143, 2015.
42. Markus N Rabe, Christoph M Wintersteiger, Hillel Kugler, Boyan Yordanov, and Youssef Hamadi. Symbolic approximation of the bounded reachability probability in large Markov chains. In *Proceedings of QEST*, pages 388–403. Springer, 2014.

43. J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, January 1965.
44. Ashish Sabharwal, Carlos Ansotegui, Carla P Gomes, Justin W Hart, and Bart Selman. QBF modeling: Exploiting player symmetry for simplicity and efficiency. In *Proceedings of SAT*, pages 382–395. Springer, 2006.
45. João P Marques Silva and Karem A Sakallah. GRASP - A new search algorithm for satisfiability. In *Proceedings of CAD*, pages 220–227. IEEE, 1997.
46. Armando Solar-Lezama. *Program synthesis by sketching.* PhD thesis, University of California, Berkeley, 2008.
47. Grigori S Tseitin. On the complexity of derivation in propositional calculus. *Studies in constructive mathematics and mathematical logic*, 2(115-125):10–13, 1968.
48. Christoph M Wintersteiger, Youssef Hamadi, and Leonardo De Moura. Efficiently solving quantified bit-vector formulas. *Proceedings of FMSD*, 42(1):3–23, 2013.

## Appendix

## A Proof of Theorem 1

*Proof.* The algorithm maintains the invariant that $\forall X.\exists Y.\varphi$ is $D$-consistent. It starts with an empty set $D$, for which any 2QBF in PCNF is $D$-consistent. The procedure Propagation only adds variables to $D$ after they have been identified to be locally (and thus globally by Lemma 1) deterministic and passed the conflict checks (by Lemma 3). The procedure BackTrack reverts the data structures to a previous $D$-consistent state. When we add a conflict clause $c$, we know that the clause is not in $\mathcal{D}$, as we previously backtracked to a decision level smaller than the maximal decision level in $c$. The algorithm also adds clauses during decisions and by the pure literal rule, but these are not in $\mathcal{D}$ either at the time they are added.

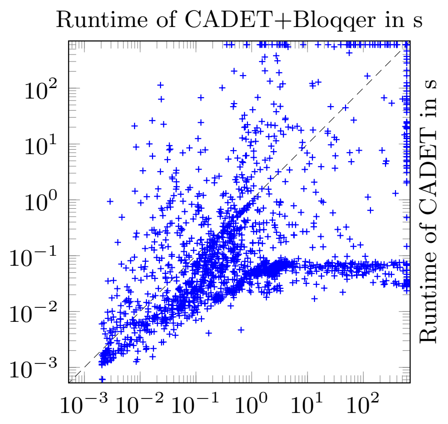
For proving termination, we first consider the propagation procedure. The size of $D$ is monotonically increasing during the execution of Propagate and is bounded by $|Y|$. Between two additions to the set $D$ (after executing line 12 until the execution of line 11), the size of $U$ is monotonically decreasing, except when the pure literal rule applies. The pure literal rule only reintroduces the variable last taken out of the set $U$ and guarantees that during the next iteration of the while loop this variable is identified to be deterministic, forcing $D$ to grow or the procedure to terminate with a conflict. The other procedures used in the algorithm trivially terminate. The number of iterations of the main loop in the algorithm IncrementalDeterminization between any two conflicts is bounded by the number of variables in $Y \setminus D$ as each decision makes $D$ grow by at least one variable. The number of conflicts is also bounded, as every conflict results in a new non-trivial clause over the variables $X$ and $D$, and there are only finitely many such clauses.

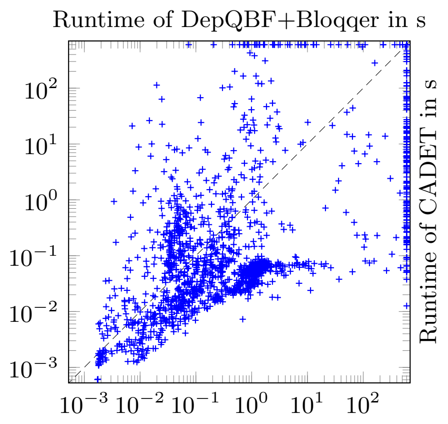
If the algorithm terminates with *true*, the formula is $Y$-consistent, thus true. The function includes the original clauses and thus cannot violate any constraints for any assignment to $X$ - the clauses represent a Skolem function satisfying all original constraints.

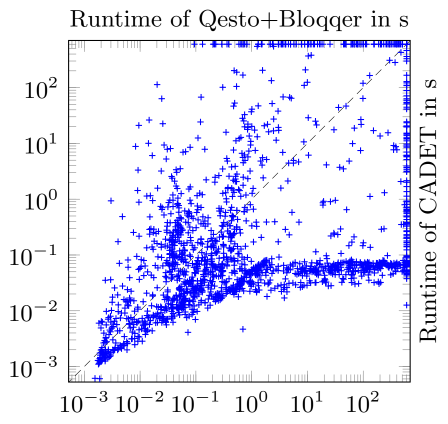
It remains to show soundness for the case that the algorithm terminates with *false*. The conflict does not depend on decision clauses and by Lemma 2 we know that the clauses introduced by the pure literal rule are sound with the original set of clauses. Lemma 4 guarantees that the conflict clauses, including the final one is sound. The final conflict clause only consists of universally quantified variables, which is equivalent to false by universal reduction. $\square$

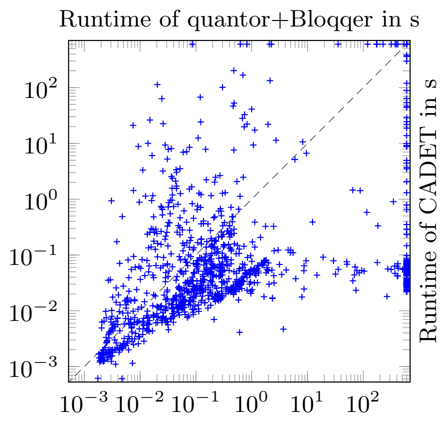
## B Supplemental Experimental Data

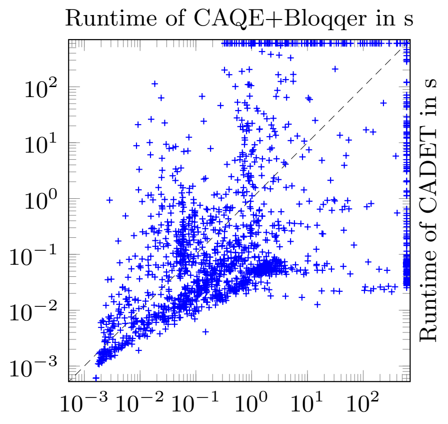
In the following we provide additional scatter plots comparing CADET to various solvers. We start with solvers that use Bloqqer as a preprocessor.

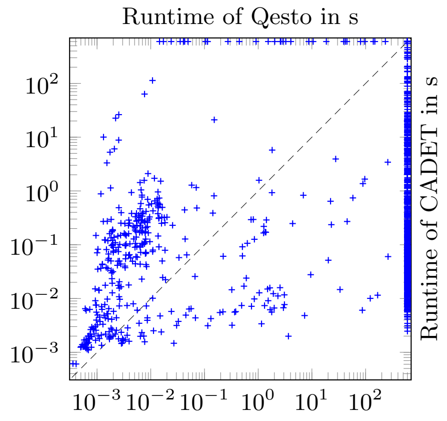

Runtime of CADET+Bloqqer in s
Runtime of CADET in s
$10^{-3}$ $10^{-2}$ $10^{-1}$ $10^{0}$ $10^{1}$ $10^{2}$

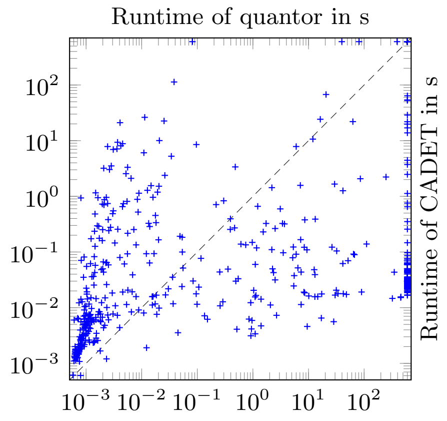

Runtime of DepQBF+Bloqqer in s
Runtime of CADET in s
$10^{-3}$ $10^{-2}$ $10^{-1}$ $10^{0}$ $10^{1}$ $10^{2}$

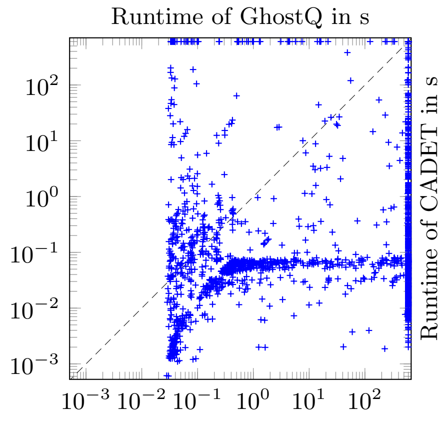

Runtime of Qesto+Bloqqer in s
Runtime of CADET in s
$10^{-3}$ $10^{-2}$ $10^{-1}$ $10^{0}$ $10^{1}$ $10^{2}$

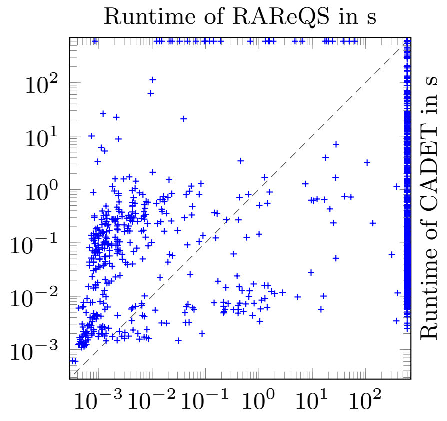

Runtime of quantor+Bloqqer in s
Runtime of CADET in s
$10^{-3}$ $10^{-2}$ $10^{-1}$ $10^{0}$ $10^{1}$ $10^{2}$

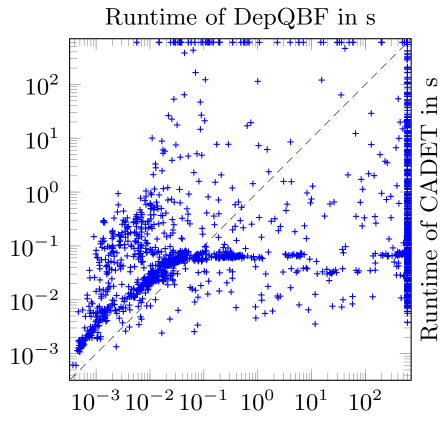

Runtime of CAQE+Bloqqer in s
Runtime of CADET in s
$10^{-3}$ $10^{-2}$ $10^{-1}$ $10^{0}$ $10^{1}$ $10^{2}$

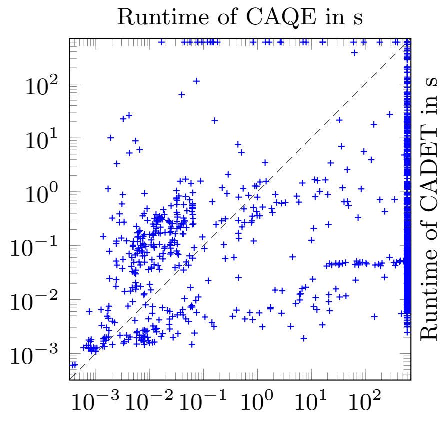












The second group of plots shows that without Bloqqer the solvers Qesto, quantor, RAReQS, and DepQBF either solve instances in less time than CADET or produce a timeout. The majority of the instances are lined up at the right fringe of the plots. An interesting observation is that GhostQ, like the CEGAR and QDPLL solvers, appears to have groups of benchmarks that are almost orthogonal in the scatter plot. Note that the upper part of the right fringe of the plot comparing quantor to CADET is much less populated than for the other plots. This is an artifact of the evaluation where instances on which a solver produced a MEMOUT are not indicated in the plot. Quantor was the only solver to produce MEMOUTs.

## C Certificates

While the primary use of certificates is to make sure that the computed result is correct, there are some applications in which the certificates are of further use as they represent implementations or strategies. It is therefore considered important to extract reasonably small certificates. Here we present more experimental data on the size of the produced certificates. Consider the scatter plot in Fig. 4, which compares the number of clauses in the formulas (that have been identified by CADET to be true) to the number of gates in the certificates provided by CADET after minimization with the ABC model checker. We see that for some benchmarks certificate sizes roughly correlate with the problem size. For the benchmarks Random and Partial Equivalence, however, this does not hold true.

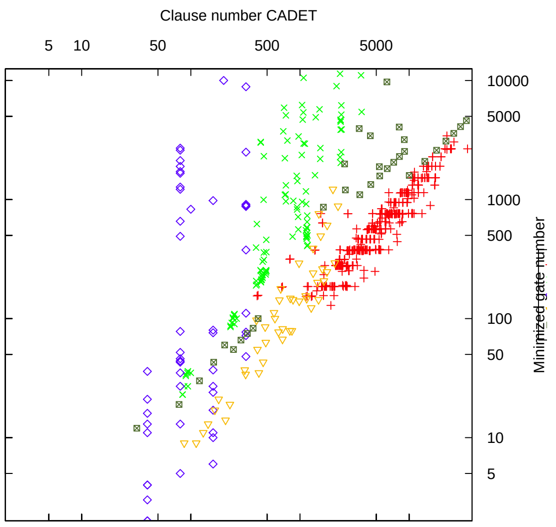


**Fig. 4.** Comparing certificate sizes (number of gates) to problem sizes (number of clauses). Diamond shapes are represent the Random benchmark, green crosses represent the Partial Equivalence benchmark, yellow triangles represent the Synthesis benchmark, grey-green boxes represent the Hardware Fixpoint benchmark, and red crosses represent the Ranking Function benchmark.

## C.1 Example Certificates

In the following we present two certificates that were computed by CADET. They are presented as Aiger circuits, which have the following elements: Blue triangles are input and output signals, square nodes are input signals, and oval nodes are And gates. The inputs are at the bottom of the plots and connections always go up. Black dots indicate that the signal is negated.

The two examples shown in Fig. 5 are taken from the Ranking Functions benchmark and the Terminator benchmark. The circuits were simplified by ABC using the command `dc2`. In both cases we cropped a number of constant outputs to fit the certificates onto the pages. We can observe that the certificates involve a relatively low number of gates. The ranking function example shows several disconnected repetitive structures that could be arithmetic operations.

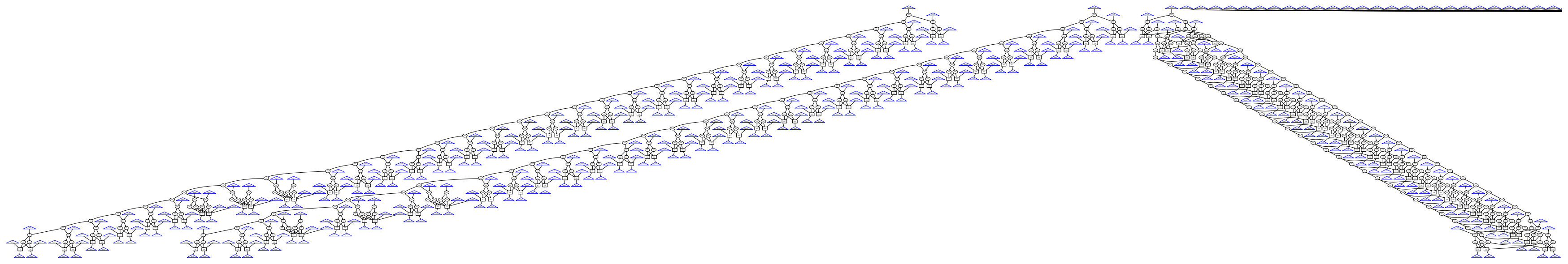
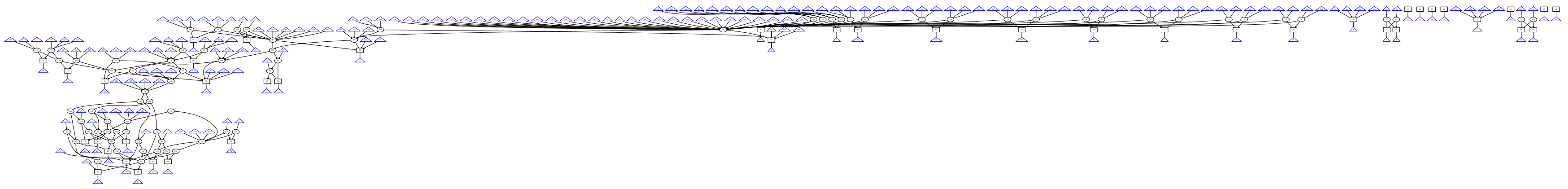



**Fig. 5.** Certificates computed by CADET for `rankfunc60_unsigned_32.qdimacs` (upper) and `stmt9_350_351.qdimacs` (lower).

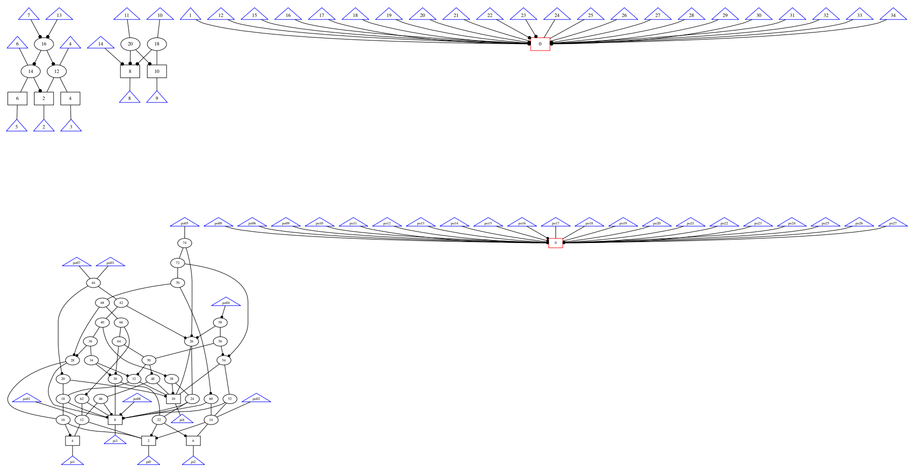


**Fig. 6.** Certificates for `stmt6_13_14.qdimacs` computed by CADET (upper, 5 gates) and DepQBF (lower, 32 gates).

## C.2 Comparison to Certificates Generated by DepQBF

A comparative evaluation is difficult, because the other certifying solvers CAQE and DepQBF terminate only on a fraction of the benchmarks that can be solved by CADET (without using Bloqqer). We nevertheless want to give an impression and compare the certificates for the instance `stmt6_13_14.qdimacs`, which has 34 variables and 76 clauses and is solved almost instantly by both CADET and DepQBF. Without simplification the CADETs certificate has 8 gates, and DepQBF's certificate has 78 gates. After simplifying the certificates using the ABC model checker they look as depicted in Figure 6.