Quantifier Elimination via Clause Redundancy

Eugene Goldberg and Panagiotis Manolios
Northeastern University, USA, {eigold,pete}@ccs.neu.edu

***Abstract*—We consider the problem of existential quantifier elimination for Boolean formulas in conjunctive normal form. Recently we presented a new method for solving this problem based on the machinery of Dependency sequents (D-sequents). The essence of this method is to add to the quantified formula implied clauses until all the clauses with quantified variables become redundant. A D-sequent is a record of the fact that a set of quantified variables is redundant in some subspace. In this paper, we introduce a quantifier elimination algorithm based on a new type of D-sequents called *clause D-sequents* that express redundancy of clauses rather than variables. Clause D-sequents significantly extend our ability to introduce and algorithmically exploit redundancy, as our experimental results show.**

I. Introduction

In this paper, we consider elimination of quantifiers from formulas of the form $\exists X[F]$ where F is a Boolean formula in conjunctive normal form (CNF). We will refer to such formulas as $\exists$CNF. The **Quantifier Elimination (QE) problem**, is to find a quantifier-free CNF formula G such that $G \equiv \exists X[F]$. The equivalence '$\equiv$' is semantic. That is for every complete assignment $\boldsymbol{s}$ to the non-quantified variables of F, the logical value of $G_{\boldsymbol{s}}$ is equal to that of $\exists X[F_{\boldsymbol{s}}]$. Here $F_{\boldsymbol{s}}$ and $G_{\boldsymbol{s}}$ are formulas F and G under assignment $\boldsymbol{s}$.

The motivation for studying the QE problem is twofold. First, a QE algorithm can be used for solving many verification problems e.g. computing reachable states [5], [17]. Second, the methods developed for QE may come handy for other problems. For example, the machinery of Dependency sequents (D-sequents) [9], [10] that we continue developing in this paper can be used for SAT-solving [8].

In [9], [10], we developed a QE algorithm called DDS (Derivation of D-Sequents) based on the following two ideas. The first idea is that adding resolvent clauses to formula F eventually makes the clauses containing a variable of X (we will call them ***X*-clauses**) redundant. Let H denote formula $F \wedge G$ where G is the set of added resolvent clauses. For the sake of convenience, since a CNF formula can be considered as a set of clauses, we will use logical and set-based notation interchangeably. For example, formula $F \wedge G$ can also be written as $F \cup G$. The redundancy of X-clauses in $\exists X[H]$ means that $\exists X[H] \equiv \exists X[H \setminus H^X]$ where H^X is the set of X-clauses of H. Since $H \setminus H^X$ does not depend on X, the quantifiers can be dropped. So the set of clauses $H \setminus H^X$ is a quantifier-free formula equivalent to $\exists X[H]$ and so to $\exists X[F]$. The second idea is to use a divide-and-conquer strategy. DDS proves redundancy of X-clauses in subspaces and then merges the results of different branches.

A successful implementation of DDS became possible only due to development of the machinery of D-sequents that was the main contribution of [9], [10]. A D-sequent is a record of the form $(\exists X[F], \boldsymbol{q}) \rightarrow Z$ where $\boldsymbol{q}$ is a partial assignment to variables of F and $Z \subseteq X$. This D-sequent says that the variables of Z are redundant in $\exists X[F]$ in subspace $\boldsymbol{q}$. The redundancy of variables of Z means redundancy of all X-clauses containing a variable of Z. For the sake of brevity, in the following exposition we use the same symbol F to denote the initial and the *current* CNF formula consisting of the initial clauses and *resolvents*. So symbol F used in the D-sequent above is the current CNF formula.

DDS keeps adding resolvent clauses to formula F until D-sequent $(\exists X[F], \emptyset) \rightarrow X$ is derived stating that the variables of X are redundant in formula $\exists X[F]$ globally. (This means that $F \setminus F^X$ is a solution to the QE problem.) The derivation of such D-sequent is achieved by generation of atomic D-sequents and using a resolution-like operation *join*. An atomic D-sequent is derived when redundancy of a variable of X in a subspace can be trivially proved. Operation *join* is applied to D-sequents $(\exists X[F], \boldsymbol{q'}) \rightarrow Z$ and $(\exists X[F], \boldsymbol{q''}) \rightarrow Z$ where $\boldsymbol{q'}$ and $\boldsymbol{q''}$ contain opposite assignments to a variable v of F. The result of *join* is a new D-sequent $(\exists X[F], \boldsymbol{q}) \rightarrow Z$ where $\boldsymbol{q}$ consists of all assignments of $\boldsymbol{q'}$,$\boldsymbol{q''}$ but those to variable v.

The main contribution of this paper is the development of the machinery of a new type of D-sequents called **clause D-sequents**. A clause D-sequent is a record of the form $(\exists X[F], \boldsymbol{q}) \rightarrow R$ where $R \subseteq F^X$. It states that the X-clauses of R are redundant in $\exists X[F]$ in subspace $\boldsymbol{q}$. Clause D-sequents can be used to express redundancy of *any* subset of X-clauses while D-sequents of [9] can do this only for *some* subsets of X-clauses. Namely, a D-sequent of [9] can express redundancy of a set $R \subseteq F^X$ *only* if R is the set of *all* X-clauses containing variables from a set Z. (To distinguish new and old D-sequents we will refer to the latter as D-sequents based on variable redundancy.) For instance, a D-sequent based on variable redundancy cannot express the fact that a single X-clause C is redundant in $\exists X[F]$ in subspace $\boldsymbol{q}$. Similarly to D-sequents based on variable redundancy, the machinery of clause D-sequents is based on a) derivation of atomic clause D-sequents, b) a resolution-like operation *join* and c) removing redundant X-clauses from the formula to guarantee the composability of D-sequents. The latter means that proving redundancy of sets of clauses R' and R'' independently implies that the set $R' \cup R''$ is also redundant.

To show the advantages of clause D-sequents we describe a new QE algorithm called ***DCDS*** (Derivation of Clause D-Sequents). Development of $DCDS$ is another contribution of this paper. $DCDS$ can be viewed as an adaptation of DDS to clause D-sequents. However, this adaptation is far from being trivial because clause D-sequents have new features that D-sequents based on variable redundancy do not. Using clause D-sequents is beneficial for at least two reasons. First, $DCDS$ has much more flexibility than DDS in proving redundancy of X-clauses. Proving that a variable $v \in X$ is redundant in $\exists X[F]$

ISBN 978-0-9835678-3-7/13.

in subspace $\boldsymbol{q}$ by DDS requires proving redundancy of all clauses of F with variable v *at the same time*. $DCDS$ has no such restriction. Some clauses with variable v may be proved redundant much later than the others. Second, the size of clause D-sequents is in general *smaller* than that of D-sequents based on variable redundancy. (The size of D-sequent $(\exists X[F], \boldsymbol{q}) \rightarrow R$ is the number of variables assigned in $\boldsymbol{q}$.) The reason is that proving redundancy of a clause is easier than that of a variable. This facilitates pruning the search space like derivation of shorter clauses does in SAT-solving.

This paper is structured as follows. In Section II, we give basic definitions. Simple cases of clause redundancy are discussed in Section III. Clause D-sequents are introduced in Section IV. Section V describes a new QE algorithm called $DCDS$. In Section VI, we explain $DCDS$ by a simple example. Experimental results are given in Section VII. Background is discussed in Section VIII, and conclusions are presented in Section IX.

II. Basic Definitions

Definition 1: An $\boldsymbol{\exists CNF\ \ formula}$ is a quantified CNF formula of the form $\exists X[F]$ where F is a CNF formula, and X is a set of Boolean variables. If we do not explicitly specify whether we are referring to CNF or $\exists$CNF formulas, when we write "formula" we mean either a CNF or $\exists$CNF formula. Let $\boldsymbol{q}$ be an assignment, F be a CNF formula, and C be a clause. $Vars(\boldsymbol{q})$ denotes the variables assigned in $\boldsymbol{q}$; $Vars(F)$ denotes the set of variables of F; $Vars(C)$ denotes the variables of C; and $Vars(\exists X[F]) = Vars(F) \setminus X$.

We consider *true* and *false* as a special kind of clauses. A non-empty clause C becomes *true* when it is satisfied by an assignment $\boldsymbol{q}$ i.e. when a literal of C is set to *true* by $\boldsymbol{q}$. A clause C becomes *false* when it is falsified by $\boldsymbol{q}$ i.e. when all the literals of C are set to *false* by $\boldsymbol{q}$.

Definition 2: Let C be a clause, H be a formula, and $\boldsymbol{q}$ be an assignment such that $Vars(\boldsymbol{q}) \subseteq Vars(H)$. Denote by $C_{\boldsymbol{q}}$ the clause equal to *true* if C is satisfied by $\boldsymbol{q}$; otherwise $C_{\boldsymbol{q}}$ is the clause obtained from C by removing all literals falsified by $\boldsymbol{q}$. $H_{\boldsymbol{q}}$ denotes the formula obtained from H by replacing every clause C of H with $C_{\boldsymbol{q}}$. In the context of this paper, it is convenient to assume that clause $C_{\boldsymbol{q}}$ equal to *true* remains in $H_{\boldsymbol{q}}$ rather than being removed from $H_{\boldsymbol{q}}$. We treat such a clause as *redundant* in $H_{\boldsymbol{q}}$ (see Proposition 1).

Definition 3: Let G, H be formulas. We say that G, H are **equivalent**, written $G \equiv H$, if for all assignments, $\boldsymbol{y}$, such that $Vars(\boldsymbol{y}) \supseteq (Vars(G) \cup Vars(H))$, we have $G_{\boldsymbol{y}} = H_{\boldsymbol{y}}$ (notice that $G_{\boldsymbol{y}}$ and $H_{\boldsymbol{y}}$ have no free variables, so by $G_{\boldsymbol{y}} = H_{\boldsymbol{y}}$ we mean semantic equivalence). Observe that if $Vars(\boldsymbol{q}) \subseteq Vars(\exists X[F])$, then $(\exists X[F])_{\boldsymbol{q}} \equiv \exists X[F_{\boldsymbol{q}}]$.

Definition 4: The **Quantifier Elimination (QE) problem** for $\exists$CNF formula $\exists X[F]$ consists of finding a CNF formula G such that $G \equiv \exists X[F]$.

Definition 5: Let Z be a set of variables. A clause C of F is called a $\boldsymbol{Z}$**-clause** if $Vars(C) \cap Z \neq \emptyset$. We denote by $\boldsymbol{F^Z}$ the set of all Z-clauses of F.

Definition 6: Let X be a set of Boolean variables, F be a CNF formula and R be a subset of X-clauses of F. The clauses of R are **redundant** in CNF formula F if $F \equiv (F \setminus R)$. The clauses of R are **redundant** in $\exists$CNF formula $\exists X[F]$ if $\exists X[F] \equiv \exists X[F \setminus R]$. Note that $F \equiv (F \setminus R)$ implies $\exists X[F] \equiv \exists X[F \setminus R]$ but the opposite is not true.

III. Simple Cases of Clause Redundancy

In this section, we describe three situations where clause redundancy can be trivially proved (Propositions 1, 2, 3).

Proposition 1: Let $\exists X[F]$ be an $\exists$CNF formula and $\boldsymbol{q}$ be an assignment to $Vars(F)$ satisfying an X-clause C of F. Then $C_{\boldsymbol{q}}$ is redundant in $\exists X[F_{\boldsymbol{q}}]$.

Due to lack of space we omit proofs. Note that proofs of non-trivial propositions of this paper are similar to those of [10] given for D-sequents based on variable redundancy.

Proposition 2: Let $\exists X[F]$ be an $\exists$CNF formula and $\boldsymbol{q}$ be an assignment to $Vars(F)$. Let C, C' be two clauses of F. Let C be falsified by $\boldsymbol{q}$ and C' be an X-clause. Then $C'_{\boldsymbol{q}}$ is redundant in $\exists X[F_{\boldsymbol{q}}]$.

To formulate Proposition 3 below, we need to introduce a few definitions.

Definition 7: Let C' and C'' be clauses having opposite literals of exactly one variable $v \in Vars(C') \cap Vars(C'')$. The clause C consisting of all literals of C' and C'' but those of v is called the **resolvent** of C',C'' on v. Clause C is said to be obtained by **resolution** on v. Clauses C',C'' are called **resolvable** on v.

Definition 8: A clause C of a CNF formula F is called **blocked** at variable v, if no clause of F is resolvable with C on v. The notion of blocked clauses was introduced in [15].

Proposition 3: Let $\exists X[F]$ be an $\exists$CNF formula and $\boldsymbol{q}$ be an assignment to $Vars(F)$. Let C be an X-clause of F not satisfied by $\boldsymbol{q}$ and $v \in X$ be a variable of C such that $v \notin Vars(\boldsymbol{q})$. Let clause $C_{\boldsymbol{q}}$ be blocked at v in $F_{\boldsymbol{q}}$. Then $C_{\boldsymbol{q}}$ is redundant in $\exists X[F_{\boldsymbol{q}}]$.

IV. Dependency Sequents Based on Clause Redundancy

In this section, we define a new kind of dependency sequents (D-sequents) called clause D-sequents. In contrast to D-sequents of [9], clause D-sequents are based on the notion of clause redundancy. We describe operation *join* producing a new clause D-sequent from existing ones and introduce the notion of composable clause D-sequents.

A. Definition of D-sequents

Definition 9: Let $\exists X[F]$ be an $\exists$CNF formula. Let $\boldsymbol{q}$ be an assignment to $Vars(F)$ and R be a subset of X-clauses of F. A clause dependency sequent (**clause D-sequent**) has the form $(\exists X[F], \boldsymbol{q}) \rightarrow R$. It states that the clauses of $R_{\boldsymbol{q}}$ are redundant in $\exists X[F_{\boldsymbol{q}}]$.

From now on we will refer to clause D-sequents as *just* D-sequents unless we want to contrast clause D-sequents with those based on variable redundancy.

Example 1: Consider an $\exists$CNF formula $\exists X[F]$ where $F = C_1 \wedge C_2$, $C_1 = x \vee y_1$ and $C_2 = \overline{x} \vee y_2$ and $X = \{x\}$. Let

$\boldsymbol{q}$=$\{(y_1 = 1)\}$. Then clause C_1 is satisfied by $\boldsymbol{q}$ and according to Proposition 1, the D-sequent $(\exists x[F], \boldsymbol{q}) \rightarrow \{C_1\}$ holds. Since $F_{\boldsymbol{q}}$=$\{true, C_2\}$, clause C_2 is blocked at variable x. So according to Proposition 3, the D-sequent $(\exists x[F], \boldsymbol{q}) \rightarrow \{C_2\}$ holds.

According to Definition 9, a D-sequent holds with respect to a *particular* $\exists$CNF formula $\exists X[F]$. Proposition 4 below shows that this D-sequent also holds after adding to F any set of resolvent clauses.

Proposition 4: Let $\exists X[F]$ be an $\exists$CNF formula. Let $\boldsymbol{q}$ be an assignment to $Vars(F)$. Let G be a CNF formula such that $F \Rightarrow G$. Then if $(\exists X[F], \boldsymbol{q}) \rightarrow R$ holds, $(\exists X[F \wedge G], \boldsymbol{q}) \rightarrow R$ does too.

B. Join Operation for D-sequents

In this subsection, we introduce the operation of joining D-sequents (Definition 11).

Definition 10: Let $\boldsymbol{q'}$ and $\boldsymbol{q''}$ be assignments in which exactly one variable $v \in Vars(\boldsymbol{q'}) \cap Vars(\boldsymbol{q''})$ is assigned different values. The assignment $\boldsymbol{q}$ consisting of all the assignments of $\boldsymbol{q'}$ and $\boldsymbol{q''}$ but those to v is called the **resolvent** of $\boldsymbol{q'}$,$\boldsymbol{q''}$ on v. Assignments $\boldsymbol{q'}$,$\boldsymbol{q''}$ are called **resolvable** on v.

Proposition 5: Let $\exists X[F]$ be an $\exists$CNF formula. Let D-sequents $(\exists X[F], \boldsymbol{q'}) \rightarrow R$ and $(\exists X[F], \boldsymbol{q''}) \rightarrow R$ hold. Let $\boldsymbol{q'}$, $\boldsymbol{q''}$ be resolvable on $v \in Vars(F)$ and $\boldsymbol{q}$ be the resolvent of $\boldsymbol{q'}$ and $\boldsymbol{q''}$. Then, D-sequent $(\exists X[F], \boldsymbol{q}) \rightarrow R$ holds too.

Definition 11: We will say that the D-sequent $(\exists X[F], \boldsymbol{q}) \rightarrow R$ of Proposition 5 is produced by **joining D-sequents** $(\exists X[F], \boldsymbol{q'}) \rightarrow R$ and $(\exists X[F], \boldsymbol{q''}) \rightarrow R$ at v.

C. Composable D-sequents

In general, the fact that D-sequents $(\exists X[F], \boldsymbol{q}) \rightarrow R'$ and $(\exists X[F], \boldsymbol{q}) \rightarrow R''$ hold does not imply that $(\exists X[F], \boldsymbol{q}) \rightarrow R' \cup R''$ holds. The reason is that redundancy of R' may be true only when clauses of R'' are in F and vice versa. So, derivation of $(\exists X[F], \boldsymbol{q}) \rightarrow R' \cup R''$ requires recursive reasoning. Proposition 6 below shows how to avoid recursive reasoning.

Let $\boldsymbol{q}$ and $\boldsymbol{s}$ be assignments to a set of variables Z. Since $\boldsymbol{q}$ and $\boldsymbol{s}$ are sets of value assignments to individual variables of Z, one can apply set operations to them. For instance, $\boldsymbol{s} \subseteq \boldsymbol{q}$ means that $\boldsymbol{q}$ contains the value assignments of $\boldsymbol{s}$. Assignment $\boldsymbol{q} \cup \boldsymbol{s}$ consists of the value assignments that are in $\boldsymbol{q}$ or $\boldsymbol{s}$.

Proposition 6: Let $\boldsymbol{s}$ and $\boldsymbol{q}$ be assignments to variables of F where $\boldsymbol{s} \subseteq \boldsymbol{q}$. Let D-sequents $(\exists X[F], \boldsymbol{s}) \rightarrow R'$ and $(\exists X[F \setminus R'], \boldsymbol{q}) \rightarrow R''$ hold. Then D-sequent $(\exists X[F], \boldsymbol{q}) \rightarrow R' \cup R''$ holds.

Definition 12: Let $\boldsymbol{q'}$ and $\boldsymbol{q''}$ be assignments to a set of variables Z. We will say that $\boldsymbol{q'}$ and $\boldsymbol{q''}$ are **compatible** if every variable of $Vars(\boldsymbol{q'}) \cap Vars(\boldsymbol{q''})$ is assigned the same value in $\boldsymbol{q'}$ and $\boldsymbol{q''}$.

Definition 13: Let D-sequent S' be equal to $(\exists X[F], \boldsymbol{q'}) \rightarrow R'$ and S'' be equal to $(\exists X[F], \boldsymbol{q''}) \rightarrow R''$ where $\boldsymbol{q'}$ and $\boldsymbol{q''}$ are compatible assignments to $Vars(F)$. We will call S' and S'' **composable** if D-sequent S equal to $(\exists X[F], \boldsymbol{q'} \cup \boldsymbol{q''}) \rightarrow R' \cup R''$ holds. From Proposition 6 it follows that S', S'' are composable if D-sequent $(\exists X[F \setminus R'], \boldsymbol{q'} \cup \boldsymbol{q''}) \rightarrow R''$ or $(\exists X[F \setminus R''], \boldsymbol{q'} \cup \boldsymbol{q''}) \rightarrow R'$ hold.

Although the QE algorithm of [9] derives composable D-sequents we did not explicitly use the notion of composability of D-sequents there. In this paper, we make this important notion more conspicuous.

```
// q is an assignment to Vars(F)
// Ω denotes a set of active D-sequents
// Φ denotes ∃X[F]
// If DCDS returns clause nil (respectively a non-nil clause),
// F_q is satisfiable (respectively unsatisfiable)

DCDS(Φ,q,Ω){
1   if (∃ clause C ∈ F falsif. by q) {
2     Ω := atomic_Dseqs1(Ω, q, C);
3     return(Φ, Ω, C);}
4   Ω := atomic_Dseqs2(Φ, q, Ω);
5   if (all_X_clauses_redund(Φ, Ω)) return(Φ, Ω, nil);
    - - - - - - - - - - - -
6   v := pick_variable(F, q, Ω);
7   (Φ, Ω, C_0) :=DCDS(Φ,q ∪ (v = 0),Ω);
8   if (C_0 ≠ nil) Ω := add_atomic_Dseqs(Ω, q, C_0);
9   Ω_asym := Dseqs_to_be_inactive(F, Ω, v);
10  if (Ω_asym = ∅) return(Φ, Ω, C_0);
11  Ω := Ω \ Ω_asym;
12  (Φ, Ω, C_1) :=DCDS(Φ,q ∪ (v = 1),Ω);
    - - - - - - - - - - - -
13  if ((C_0 ≠ nil) and (C_1 ≠ nil)){
14    C := resolve_clauses(C_0, C_1, v);
15    F := F ∧ C;
16    Ω := atomic_Dseqs1(Ω, q, C);
17    return(Φ, Ω, C);}
18  Ω := merge(Φ, q, v, Ω_asym, Ω, C_0, C_1);
19  return(Φ, Ω, nil);}
```

Fig. 1. $DCDS$ procedure

V. Algorithm Description

In this section, we describe a QE algorithm called ***DCDS*** (Derivation of Clause D-Sequents). $DCDS$ derives D-sequents $(\exists X[F], \boldsymbol{q}) \rightarrow \{C\}$ stating the redundancy of X-clause $C_{\boldsymbol{q}}$ of $F_{\boldsymbol{q}}$. From now on, we will use a short notation of D-sequents writing $\boldsymbol{s} \rightarrow \{C\}$ instead of $(\exists X[F], \boldsymbol{s}) \rightarrow \{C\}$. We will assume that the parameter $\exists X[F]$ missing in $\boldsymbol{s} \rightarrow \{C\}$ is the *current* $\exists$CNF formula (with all resolvents added to F).

One can omit $\exists X[F]$ from D-sequents because from Proposition 4 it follows that $(\exists X[F], \boldsymbol{s}) \rightarrow \{C\}$ holds no matter how many resolvent clauses are added to F. We will call D-sequent $\boldsymbol{s} \rightarrow \{C\}$ **active** in subspace $\boldsymbol{q}$ if $\boldsymbol{s} \subseteq \boldsymbol{q}$. If $\boldsymbol{s} \rightarrow \{C\}$ is active in subspace $\boldsymbol{q}$, clause $C_{\boldsymbol{q}}$ is redundant in $\exists X[F_{\boldsymbol{q}}]$.

A description of $DCDS$ is given in Figure 1. $DCDS$ accepts an $\exists$CNF formula $\exists X[F]$ (denoted as Φ), an assignment $\boldsymbol{q}$ to $Vars(F)$ and a set Ω of D-sequents active in subspace $\boldsymbol{q}$ stating redundancy of *some* X-clauses in $\exists X[F_{\boldsymbol{q}}]$. To simplify description of $DCDS$, by ***X*-clauses of $F_{\boldsymbol{q}}$** we also mean the X-clauses of F satisfied by $\boldsymbol{q}$. On the contrary, an X-clause of F falsified by $\boldsymbol{q}$ is not considered as an X-clause of $F_{\boldsymbol{q}}$. $DCDS$ returns a formula $\exists X[F]$ modified by resolvent clauses added to F (if any), a set Ω of D-sequents active in subspace $\boldsymbol{q}$ that state redundancy of *all* X-clauses in $\exists X[F_{\boldsymbol{q}}]$

and a clause C. If $F_{\boldsymbol{q}}$ is unsatisfiable then C is a clause of F falsified by $\boldsymbol{q}$. Otherwise, C is equal to *nil* meaning that no clause implied by F is falsified by $\boldsymbol{q}$.

The active D-sequents derived by $DCDS$ are composable. That is if $\boldsymbol{s_1} \to \{C_1\}, \ldots, \boldsymbol{s_k} \to \{C_k\}$ are the active D-sequents of subspace $\boldsymbol{q}$, then the D-sequent $\boldsymbol{s}^* \to \{C_1, \ldots, C_k\}$ holds where $\boldsymbol{s}^* = \boldsymbol{s_1} \cup \ldots \cup \boldsymbol{s_k}$ and $\boldsymbol{s}^* \subseteq \boldsymbol{q}$. $DCDS$ achieves composability of D-sequents as follows. As soon as an X-clause $C_{\boldsymbol{q}}$ is proved redundant, it is marked and ignored by $DCDS$, which is equivalent to removing $C_{\boldsymbol{q}}$ from $F_{\boldsymbol{q}}$. So $DCDS$ guarantees that for every path of the search tree leading to a leaf, X-clauses are proved redundant in a particular order. (This order may be different for different paths.) This allows to avoid recursive reasoning where a clause $C'_{\boldsymbol{q}}$ is used to prove redundancy of clause $C''_{\boldsymbol{q}}$ and vice versa. In turn, avoiding recursive reasoning guarantees composability of D-sequents.

A solution to the QE problem in subspace $\boldsymbol{q}$ is obtained by discarding all X-clauses from the CNF formula $F_{\boldsymbol{q}}$ of $\exists X[F_{\boldsymbol{q}}]$ returned by $DCDS$. To build a quantifier-free CNF formula equivalent to Φ, one needs to call $DCDS$ with $\boldsymbol{q} = \emptyset$, $\Omega = \emptyset$.

A. The Big Picture

$DCDS$ consists of three parts separated in Figure 1 by dotted lines. In the first part (lines 1-5), $DCDS$ builds atomic D-sequents i.e. D-sequents for X-clauses whose redundancy can be trivially proved. If all X-clauses are proved redundant in $\exists X[F_{\boldsymbol{q}}]$, $DCDS$ terminates.

If some X-clauses are not proved redundant yet, $DCDS$ enters the second part of the code (lines 6-12). First, $DCDS$ picks a branching variable v (line 6). Then it extends $\boldsymbol{q}$ by assignment to variable v and recursively calls itself (line 7) starting the left branch of v. For the sake of clarity, we assume that $DCDS$ first explores assignment $v = 0$. Once the left branch is finished, $DCDS$ extends $\boldsymbol{q}$ by $(v = 1)$ and explores the right branch (line 12).

In the third part, $DCDS$ merges the left and right branches (lines 13-19). The result of this merging is proving every X-clause redundant in $\exists X[F_{\boldsymbol{q}}]$. For every X-clause $C_{\boldsymbol{q}}$ proved redundant in $\exists X[F_{\boldsymbol{q}}]$, the set Ω contains precisely one active D-sequent $\boldsymbol{s} \to \{C\}$ where $\boldsymbol{s} \subseteq \boldsymbol{q}$. As soon as $C_{\boldsymbol{q}}$ is proved redundant, it is marked and ignored until $DCDS$ enters a subspace $\boldsymbol{q'}$ where $\boldsymbol{s} \not\subseteq \boldsymbol{q'}$ i.e. a subspace where D-sequent $\boldsymbol{s} \to \{C\}$ becomes inactive. Clause $C_{\boldsymbol{q'}}$ is unmarked in $F_{\boldsymbol{q'}}$ signaling that $DCDS$ needs to derive a new D-sequent $\boldsymbol{s'} \to \{C\}$ where $\boldsymbol{s'} \subseteq \boldsymbol{q'}$ stating the redundancy of $C_{\boldsymbol{q'}}$.

B. Building Atomic D-sequents

Procedures *atomic_Dseqs1* and *atomic_Dseqs2* are called by $DCDS$ to compute D-sequents for trivial cases of clause redundancy listed in Section III. We refer to such D-sequents as **atomic**. Procedure *atomic_Dseqs1* is called when formula $F_{\boldsymbol{q}}$ contains an empty clause $C_{\boldsymbol{q}}$ which means that clause C of F is falsified by $\boldsymbol{q}$. For every X-clause $C'_{\boldsymbol{q}}$ of $F_{\boldsymbol{q}}$ that has no active D-sequent yet, *atomic_Dseq1* generates a D-sequent $\boldsymbol{s} \to \{C'\}$. Here $\boldsymbol{s}$ is the shortest assignment falsifying C.

$atomic\_Dseqs2(\Phi, \boldsymbol{q}, \Omega)\{$
1 $\Omega := \Omega \cup Dseqs(new\_satisf\_clauses(\Phi, \boldsymbol{q}, \Omega))$;
2 $\Omega := \Omega \cup Dseqs(new\_blocked\_clauses(\Phi, \boldsymbol{q}, \Omega))$;
3 return(Ω);

Fig. 2. *atomic_Dseqs2* procedure

If $F_{\boldsymbol{q}}$ does not have an empty clause, procedure *atomic_Dseqs2* shown in Figure 2 is called. It builds D-sequents for X-clauses that became satisfied or blocked in $F_{\boldsymbol{q}}$. Let C be a clause satisfied by $\boldsymbol{q}$. Then D-sequent $\boldsymbol{s} \to \{C\}$ is generated where $\boldsymbol{s} = (v = b)$, $b \in \{0, 1\}$ is the assignment to a variable v satisfying C.

Let $C_{\boldsymbol{q}}$ be blocked in $F_{\boldsymbol{q}}$ at variable $v \in X$ that is not assigned yet. A D-sequent $\boldsymbol{s} \to \{C\}$ stating redundancy of C is built as follows. The fact that $C_{\boldsymbol{q}}$ is blocked at v means that every clause C' of F resolvable with C on v is either satisfied by $\boldsymbol{q}$ or $C'_{\boldsymbol{q}}$ is proved redundant in $F_{\boldsymbol{q}}$. The assignment $\boldsymbol{s}$ is a subset of assignments of $\boldsymbol{q}$ guaranteeing that C' remains satisfied by $\boldsymbol{s}$ or $C'_{\boldsymbol{s}}$ remains redundant in $\exists X[F_{\boldsymbol{s}}]$ and so $C_{\boldsymbol{s}}$ is blocked at v in $F_{\boldsymbol{s}}$. If C' is satisfied by $\boldsymbol{q}$, then $\boldsymbol{s}$ contains an assignment $(v = b)$, $b \in \{0, 1\}$ of $\boldsymbol{q}$ satisfying C'. If C' is not satisfied but $C'_{\boldsymbol{q}}$ is proved redundant in $\exists X[F_{\boldsymbol{q}}]$, then $\boldsymbol{s}$ contains *all* assignments of $\boldsymbol{s'}$ where $\boldsymbol{s'} \subseteq \boldsymbol{q}$ and $\boldsymbol{s'} \to \{C'\}$ is the D-sequent of Ω stating redundancy of $C'_{\boldsymbol{q}}$.

C. Selection of a Branching Variable

Let $\boldsymbol{q}$ be the assignment $DCDS$ is called with. We will say that a **variable** x of X **is redundant** in $\exists X[F_{\boldsymbol{q}}]$ if x is not assigned in $\boldsymbol{q}$ and every $\{x\}$-clause is proved redundant in $\exists X[F_{\boldsymbol{q}}]$. Denote by X_{red} the variables proved redundant in $\exists X[F_{\boldsymbol{q}}]$. Let $Y = Vars(F) \setminus X$. $DCDS$ branches on free (*i.e.*, unassigned) variables of X and Y. Importantly, a free variable $x \in X \setminus Vars(\boldsymbol{q})$ is picked for branching *only* if $x \notin X_{red}$ i.e. $DCDS$ does not branch on variables proved redundant.

Although Boolean Constraint Propagation (BCP) is not shown explicitly in Figure 1, it is included into the *pick_variable* procedure as follows: a) preference is given to branching on variables of unit clauses of $F_{\boldsymbol{q}}$ (if any); b) if v is a variable of a unit clause $C_{\boldsymbol{q}}$ of $F_{\boldsymbol{q}}$ and v is picked for branching, then the value falsifying $C_{\boldsymbol{q}}$ is assigned first to cause immediate termination of this branch.

To simplify merging results of the left and right branches, $DCDS$ first assigns values to variables of Y (more details are given in Subsection V-E). This means that *pick_variable* never selects a variable $x \in X$ for branching, if there is a free variable of Y. In particular, BCP does not assign values to variables of X if a variable of Y is still unassigned.

D. Switching from Left to Right Branch

$DCDS$ prunes big chunks of the search space by not branching on redundant variables of X. One more powerful pruning technique of $DCDS$ discussed in this subsection is reducing the size of right branches.

Let $\boldsymbol{s} \to \{C\}$ be a D-sequent of the set Ω computed by $DCDS$ in the left branch $v = 0$ (line 7 of Figure 1). We will call this D-sequent **symmetric in** $\boldsymbol{v}$, if v is not assigned in $\boldsymbol{s}$. Otherwise, this D-sequent is called **asymmetric in** $\boldsymbol{v}$. Notice that if $\boldsymbol{s}$ is symmetric in v, the D-sequent $\boldsymbol{s} \to \{C\}$ is active in the right branch and so $C_{\boldsymbol{q_1}}$ is redundant in $\exists X[F_{\boldsymbol{q_1}}]$ where

$merge(\Phi, \boldsymbol{q}, v, \Omega_{asym}, \Omega, C_0, C_1)\{$
1 $\Omega := join\_Dseqs\_of\_old\_clauses(v, \Omega_{asym}, \Omega)$;
2 $\Omega := update\_Dseqs\_of\_new\_clauses(v, \Omega)$;
3 if $(v \notin X)$ return(Ω);
4 if $(C_0 \neq nil)$ $\Omega := \Omega \cup \{Dseq(C_0)\}$;
5 if $(C_1 \neq nil)$ $\Omega := \Omega \cup \{Dseq(C_1)\}$;
6 return(Ω);}

Fig. 3. *merge* procedure

$\boldsymbol{q_1} = \boldsymbol{q} \cup \{(v = 1)\}$. Denote by Ω_{asym} the subset of active D-sequents that are asymmetric in v. It is computed in line 9. Before exploring the right branch (line 12), the X-clauses of F whose redundancy is stated by D-sequents of Ω_{asym} become non-redundant again. So the set of X-clauses to be considered in the right branch reduces to *only* those with D-sequents from Ω_{asym}. This allows to prune big parts of the search space. In particular, if Ω_{asym} is empty there is no need to explore the right branch. In this case, $DCDS$ just returns the results of the left branch (line 10). Pruning the right branch when Ω_{asym} is empty is similar to non-chronological backtracking well known in SAT-solving [16].

E. Branch Merging

Let $\boldsymbol{q_0} = \boldsymbol{q} \cup \{(v = 0)\}$ and $\boldsymbol{q_1} = \boldsymbol{q} \cup \{(v = 1)\}$. The goal of branch merging is to use solutions of the QE problem in subspaces $\boldsymbol{q_0}$ and $\boldsymbol{q_1}$ to produce a solution to the QE problem in subspace $\boldsymbol{q}$. If both $F_{\boldsymbol{q_0}}$ and $F_{\boldsymbol{q_1}}$ are unsatisfiable, this is done as described in lines 14-17 of Figure 1. In this case, the empty clauses $(C_0)_{\boldsymbol{q_0}}$ and $(C_1)_{\boldsymbol{q_1}}$ where C_0, C_1 are clauses returned in the left and right branches respectively are solutions to the QE in subspaces $\boldsymbol{q_0}$ and $\boldsymbol{q_1}$. The empty clause $C_{\boldsymbol{q}}$ where C is the resolvent of C_0 and C_1 added to F (line 15) is a solution to the QE problem in subspace $\boldsymbol{q}$. If, say, $v \notin Vars(C_1)$ and so C_1 cannot be resolved on v, *resolve_clauses* (line 14) returns C_1 itself since C_1 is falsified by $\boldsymbol{q}$. In this case, no new clause is added to F. After C is added, *atomic_Dseqs1* completes Ω by generation of atomic D-sequents built due to presence of a clause falsified by $\boldsymbol{q}$.

Suppose that at least one of formulas $F_{\boldsymbol{q_0}}$ and $F_{\boldsymbol{q_1}}$ is satisfiable. In this case, to finish solving the QE problem in subspace $\boldsymbol{q}$, one needs to make sure that every X-clause is proved redundant in $F_{\boldsymbol{q}}$. This means that every X-clause should have a D-sequent active in subspace $\boldsymbol{q}$ and hence symmetric in the branching variable v. This work is done by procedure *merge* shown in Figure 3 that consists of three steps. In the first step, *merge* takes care of D-sequents of "old" X-clauses that is the clauses that were present in F at the time the value of v was flipped from 0 to 1. For every such X-clause, a D-sequent was derived in the left branch $v = 0$. In the second step, *merge* processes new X-clauses that is X-clauses generated in the right branch $v = 1$. No D-sequents were derived for such clauses in the branch $v = 0$. In the third step, if, say, clause C_0 mentioned above is not equal to *nil*, a D-sequent is generated for C_0 if it is an X-clause.

In the first step, *merge* needs to update only D-sequents of X-clauses that became non-redundant in the right branch because their D-sequents got inactive there (such D-sequents form set Ω_{asym}, see Subsection V-D). Let us denote this set of clauses as G. If a D-sequent of an X-clause C from G returned in the *right* branch is asymmetric in v, then *join_Dseqs_of_old_clauses* (line 1) replaces it with a D-sequent symmetric in v as follows. Let S_0 and S_1 be the D-sequents derived in the left and right branches respectively that state the redundancy of $C_{\boldsymbol{q_0}}$ and $C_{\boldsymbol{q_1}}$. Then *join_Dseqs_of_old_clauses* joins S_0 and S_1 at v producing a new D-sequent S. The latter states the redundancy of $C_{\boldsymbol{q}}$ and is symmetric in v. D-sequent S_1 is replaced in Ω with S.

Let S_1 be symmetric in v. If $F_{\boldsymbol{q_0}}$ was unsatisfiable, then S_1 remains untouched. Otherwise, *join_Dseqs_of_old_clauses* does the following. Let S_1 be equal to $\boldsymbol{s} \rightarrow \{C\}$. First, the right branch assignment $v = 1$ is added to $\boldsymbol{s}$, which makes S_1 asymmetric in v. Then S_1 is joined with S_0 at v to produce a new D-sequent S that is symmetric in v. S replaces S_1 in Ω. The reason one cannot simply keep S_1 in Ω untouched is as follows. As we mentioned above, the composability of D-sequents built by $DCDS$ is based on the assumption that for every path of the search tree, X-clauses are proved redundant in a particular order. It can be shown that using D-sequent S_1 in subspace $\boldsymbol{q}$ may violate this assumption and so break the composability of D-sequents.

Let S be a D-sequent $\boldsymbol{s} \rightarrow \{C\}$ derived in the right branch $v = 1$ where C was generated in this branch i.e. C is a new clause. Such D-sequents are processed in the second step of *merge* by procedure *update_dseqs_of_new_clauses* (line 2). If S is symmetric in v, it simply remains in Ω untouched. Otherwise, S is updated by removing the assignment to v from $\boldsymbol{s}$. One can do this because the clause C is implied by F and has never been used in the left branch. So it can be considered as proved redundant in the left branch.

Finally, *merge* performs the third step (lines 3-5). Notice that if v is not in X, then C_0 or C_1 is not an X-clause. This is because $DCDS$ assigns non-quantified variables before those of X (see Subsection V-C). So the last variable assigned in an X-clause is always a variable of X. Let us assume that $v \in X$ and $C_0 \neq nil$. (In the case $C_1 \neq nil$, *merge* works similarly.) Clause $(C_0)_{\boldsymbol{q}}$ is equal to the unit clause v. Notice that $F_{\boldsymbol{q}}$ does not contain a clause with literal $\overline{v}$ because this would mean that both $F_{\boldsymbol{q_0}}$ and $F_{\boldsymbol{q_1}}$ were unsatisfiable. So, $(C_0)_{\boldsymbol{q}}$ is blocked in $F_{\boldsymbol{q}}$ at variable v. Then an atomic D-sequent is built for C_0 as described in Subsection V-B.

F. Correctness of DCDS

Let $DCDS$ be called on formula $\Phi = \exists X[F]$ with $\boldsymbol{q} = \emptyset$ and $\Omega = \emptyset$. Here is an informal explanation of why $DCDS$ produces the correct result. First, new clauses of F are produced by resolution and so are correct in the sense they are implied by F. In particular, if F is unsatisfiable, $DCDS$ returns an empty clause that is a correct solution to the QE problem. Second, the atomic D-sequents built by $DCDS$ are correct. Third, new D-sequents produced by operation *join* are correct. Fourth, the D-sequents of individual clauses are composable. So when $DCDS$ returns to the root node of the search tree, it derives the correct D-sequent $(\exists X[F], \emptyset) \rightarrow F^X$. Hence, by removing X-clauses from F, one obtains a CNF formula that is a correct solution to the QE problem.

Proposition 7: $DCDS$ is sound and complete.

VI. A Run of $DCDS$ on a Simple Formula

Let $\exists X[F]$ be an $\exists$CNF formula where $F = C_1 \wedge C_2$, $C_1 = \overline{y}_1 \vee \overline{x}$, $C_2 = y_2 \vee x$ and $X = \{x\}$. To identify a particular

$DCDS$ call we will use the corresponding assignment $\boldsymbol{q}$. For example, $DCDS_{(y_1=1,y_2=0)}$ means that the assignments $y_1 = 1$ and $y_2 = 0$ were made at recursion depths 0 and 1 respectively. Originally, assignment $\boldsymbol{q}$ is empty, so the initial call is $DCDS_{(\emptyset)}$. Figures 4, 5 show the work of $DCDS$. We use them to explain the algorithm of $DCDS$. For the sake of simplicity, in this example, we say that clause C_i, $i = 1, 2$ is blocked/redundant in $F_{\boldsymbol{q}}$ meaning that it is clause $(C_i)_{\boldsymbol{q}}$ that is blocked/redundant in $F_{\boldsymbol{q}}$.

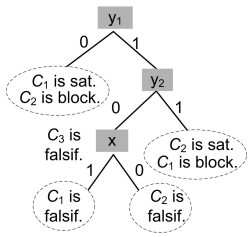

Fig. 4. Search tree built by $DCDS$

Branching variables. Figure 4 shows the search tree built by $DCDS$. Recall that $DCDS$ branches on variables of $Vars(F) \setminus X = \{y_1, y_2\}$ before those of X (see Subsection V-C).

Leaves. The search tree of Figure 4 has four leaf nodes shown in dotted ovals. In each leaf node, either the X-clauses C_1, C_2 are proved redundant or one of them is falsified. For example, C_1 is satisfied and C_2 is blocked, and hence C_1, C_2 are redundant, in leaf $(y_1 = 0)$ and clause C_1 is falsified in leaf $(y_1 = 1, y_2 = 0, x = 1)$.

Generation of new clauses. $DCDS_{(y_1=1,y_2=0)}$ generates a new clause after branching on x. $DCDS_{(y_1=1,y_2=0,x=1)}$ returns C_1 since it is falsified in $F_{(y_1=1,y_2=0,x=1)}$. Similarly, $DCDS_{(y_1=1,y_2=0,x=0)}$ returns C_2 since it is falsified in $F_{(y_1=1,y_2=0,x=0)}$. As described in Subsection V-E, in this case, $DCDS$ resolves clauses C_1 and C_2 on the branching variable x. The resolvent $C_3 = \overline{y}_1 \vee y_2$ is added to F.

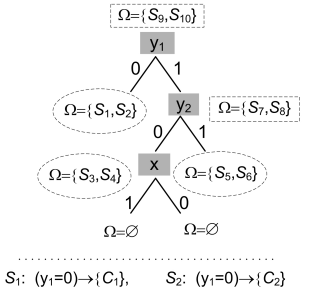

Fig. 5. Derivation of D-sequents

Generation of atomic D-sequents. Figure 5 describes derivation of D-sequents. The atomic D-sequents are shown in dotted ovals. (Dotted boxes show D-sequents obtained by operation *join*.) For instance, C_1 is satisfied by assignment $(y_1 = 0)$ and C_2 is blocked in $F_{(y_1=0)}$. So procedure *atomic_Dseqs2* called by $DCDS_{(y_1=0)}$ generates atomic D-sequents S_1 and S_2 equal to $(y_1 = 0) \rightarrow \{C_1\}$ and $(y_1 = 0) \rightarrow \{C_2\}$ respectively. The atomic D-sequents S_3, S_4 are derived by procedure *atomic_Dseqs1* called by $DCDS_{(y_1=1,y_2=0)}$. As we mentioned above, $DCDS_{(y_1=1,y_2=0)}$ adds clause $C_3 = \overline{y}_1 \vee y_2$ to F. This clause is empty in $F_{(y_1=1,y_2=0)}$ and so D-sequents S_3, S_4 equal to $\boldsymbol{s} \rightarrow \{C_1\}$, $\boldsymbol{s} \rightarrow \{C_2\}$ respectively are generated. Here $\boldsymbol{s} = (y_1 = 1, y_2 = 0)$ is the shortest assignment falsifying C_3.

Switching from left to right branch. Let us consider switching between branches by $DCDS_{(\emptyset)}$ where y_1 is picked for branching. The left branch of $\Omega_{(\emptyset)}$ returns D-sequents S_1, S_2 equal to $(y_1 = 0) \rightarrow \{C_1\}$ and $(y_1 = 0) \rightarrow \{C_2\}$ respectively.

Before starting the right branch $y_1 = 1$, $DCDS_{(\emptyset)}$ computes the set $\Omega^{asym}_{(\emptyset)}$ of D-sequents asymmetric in y_1. Since S_1 and S_2 contain an assignment to y_1, $\Omega^{asym}_{(\emptyset)}$=$\Omega_{(\emptyset)}$ and $DCDS_{(\emptyset)}$ removes S_1, S_2 from Ω. So, before $DCDS_{(y_1=1)}$ is called, both C_1 and C_2 become non-redundant again.

Branch merging. Consider how branch merging is performed by $DCDS_{(y_1=1)}$. In the left branch y_2=0, D-sequents S_3, S_4 are derived that are asymmetric in y_2. In the right branch y_2=1, D-sequents S_5, S_6, also asymmetric in y_2, are produced. By joining S_3 and S_6 at y_2, D-sequent S_7 equal to $(y_1 = 1) \rightarrow \{C_1\}$ is derived. By joining S_4 and S_5 at y_2, D-sequent S_8 equal to $(y_1$=1$) \rightarrow \{C_2\}$ is derived. D-sequents S_7, S_8 state redundancy of C_1, C_2 in $\exists X[F_{(y_1=1)}]$.

Termination. When $DCDS_{(\emptyset)}$ terminates, $F = C_1 \wedge C_2 \wedge C_3$ where $C_3 = \overline{y}_1 \vee y_2$ and composable D-sequents $\emptyset \rightarrow \{C_1\}$ and $\emptyset \rightarrow \{C_2\}$ are derived. By dropping the X-clauses C_1, C_2 one obtains $C_3 \equiv \exists X[C_1 \wedge C_2]$.

Concluding remarks. Due to small size of F, some features of $DCDS$ are not exposed. For instance, the clause C_3 generated by $DCDS$ is not an X-clause. In general, $DCDS$ may produce new X-clauses whose redundancy one needs to prove along with the original X-clauses. Another consequence of using a small example is that D-sequents derived in every node has the form $\boldsymbol{s'} \rightarrow \{C_1\}$ and $\boldsymbol{s''} \rightarrow \{C_2\}$ where $\boldsymbol{s'} = \boldsymbol{s''}$. For larger formulas, assignments $\boldsymbol{s}$ of active D-sequents $\boldsymbol{s} \rightarrow \{C\}$ may be vastly different for different clauses C.

VII. Experimental Results

The objective of experiments was to compare the performance of DDS and $DCDS$ on realistic examples and to give some comparison of D-sequent and BDD based algorithms. (A comparison of DDS with SAT-based QE algorithms is given in [9].) Importantly, in the current implementations of DDS and $DCDS$, D-sequents *are not re-used.* A D-sequent is discarded as soon as it is employed in a join operation.

We believe that reusing D-sequents will drastically boost the performance of both DDS and $DCDS$ like conflict clause re-using speeds up SAT-algorithms. Re-using learned clauses in SAT-solving is beneficial because their involvement in BCP leads to new forced assignments. Re-using D-sequents gives the power of making "asymmetric" decision assignments. If, say, assignment $v = 0$ makes a lot of learned D-sequents active, then this branch may be much easier to finish than branch $v = 1$. Note that a forced assignment can be viewed as *a special case* of an asymmetric decision assignment where one of the two branches terminates immediately. Making asymmetric decision assignments leads to smaller search trees and so re-using D-sequents provides new exciting possibilities. However, tapping this power needs extra research. For that reason, we report experimental results for the algorithms without D-sequent re-using.

TABLE I. *Results on examples solved by MC-DDS or MC-DCDS . The time limit is 2,000s*

model checker	*MC-BDD*	*MC-DDS*	*MC-DCDS*
#solved	193	247	**258**
#timeouts	66	12	1
time for solved by all three (s.)	9,080	11,293	**1,698**

We applied DDS and $DCDS$ to backward model checking. Our implementation was straightforward: DDS and

$DCDS$ were used to compute backward images until an initial state or a fixed point were reached. We will refer to these model checkers as *MC-DDS* and *MC-DCDS*. In experiments, we also used the BDD-based model checker incorporated into the latest version of a tool called PdTrav [22] (courtesy of Gianpiero Cabodi). We ran PdTrav in the backward model checking mode with ternary simulation turned off (as a non-BDD optimization). The other non-BDD optimizations, e.g. computation of the cone of influence, remained active because there was no way to switch them off. Since DDS and $DCDS$ maintain a single search tree, we also forced PdTrav to represent the transition relation by a monolithic BDD. (A D-sequent based QE algorithm *does not have to* build a single search tree e.g. it can employ restarts. However, using restarts requires storing and re-using D-sequents.) We will refer to PdTrav with the options above as *MC-BDD*.

In the experiments, we ran *MC-DDS* , *MC-DCDS* and *MC-BDD* on 758 benchmarks of the HWMCC-10 competition [23] with the time limit of 2,000 seconds. *MC-BDD* solved 374 benchmarks while *MC-DDS* and *MC-DCDS* solved 247 and 258 benchmarks respectively. This is not surprising taking into account the maturity of current BDD algorithms and their re-using of learned information via subgraph hashing. An important fact however is that *MC-DCDS* and even *MC-DDS* solved many problems that *MC-BDD* could not. So, in a sense, *MC-DDS* with *MC-DCDS* and *MC-BDD* favored different subsets of benchmarks.

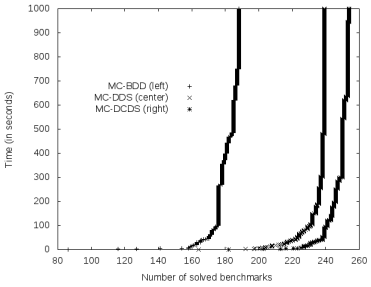

Fig. 6. Performance of model checkers on 259 examples solved by *MC-DDS* or *MC-DCDS*

Table I shows the results of the three model checkers on the 259 benchmarks solved by *MC-DDS* or *MC-DCDS* (i.e. *favored* by the two QE algorithms based on D-sequents). The second line of this table gives the number of benchmarks solved under 2,000s. The third line shows how many examples out of 259 were not solved in the time limit. The last line gives the total time in seconds for the benchmarks solved by all three model checkers. Table I shows that *MC-DCDS* significantly outperformed *MC-DDS*. Besides, a large number of problems solved by *MC-DCDS* were hard for *MC-BDD* . Figure 6 gives the performance of the three model checkers on the benchmarks solved by *MC-DDS* or *MC-DCDS* in terms of the number of problems finished in a given amount of time. *MC-DCDS* (the right line) consistently outperforms *MC-DDS* (the center line). Besides, on this set of benchmarks, both *MC-DDS* and *MC-DCDS* outperform *MC-BDD* (the left line).

Results of all three model checkers on some concrete benchmarks from the 259 benchmark set are given in Table II. Symbol '$*$' marks benchmarks that were not solved in 2,000 s. The column *iterations* show the number of backward images computed by the algorithms before finding a bug or reaching a fixed point.

TABLE II. *Some concrete examples. The time limit is 2,000s.*

benchmark	#latches	#gates	#iterations	bug	MC-BDD (s.)	MC-DDS (s.)	MC-DCDS (s.)
bj08amba4g5	36	13,637	4	no	$*$	113	16
pdtvisbakery3	48	7,514	2	yes	1.3	12	5.1
texasifetch1p5	57	663	21	yes	1.5	368	96
visprodcellp01	78	2,885	5	no	19	7.9	2.3
texasparsesysp1	312	12,173	10	yes	0.7	231	41
bobmiterbm1or	381	3,720	1	yes	0.7	0.1	0.1
pj2003	1175	15,384	3	no	$*$	$*$	252
bobsynthand	3015	15,397	2	no	1.2	0.6	0.6
mentorbm1and	4344	31,684	2	no	$*$	1.8	1.4

Table III sheds light on why $DCDS$ performs better than DDS. It shows results of applying both QE-algorithms to computing all bad states for some benchmarks (the first step of backward model checking). For either algorithm, we give the average size of atomic D-sequents and runtime. By the size of a D-sequent $\boldsymbol{s} \rightarrow \{C\}$ we mean the number of variables assigned in $\boldsymbol{s}$. To make a fair comparison we excluded the atomic D-sequents of length 1 generated when X-clauses got satisfied. Such D-sequents are not built by DDS.

TABLE III. *Relation between average size of atomic D-sequents and runtime*

benchmark	DDS		$DCDS$	
	D-seq size	time (s.)	D-seq size	time (s.)
bc57sensorsp1	19	6.3	13	0.7
boblivea	19	44	10	0.7
bobsmi2c	7.4	96	2.0	9.4
cmugigamax	20	3.6	9.4	3.2
csmacdp2	53	8.1	28	1.5
eijks344	6.0	7.5	2.5	1.6
pdtvissoap2	8.4	82	3.9	3.9
pj2006	7.3	47	1.1	0.4

The results indicate that the size of atomic D-sequents derived by $DCDS$ was smaller. The reason is as follows. When an X-clause is blocked at a variable $v \in X$, $DCDS$ generates a D-sequent $\boldsymbol{s} \rightarrow \{C\}$ where $\boldsymbol{s}$ depends *only* on clauses that can be resolved with C on a variable v. On the contrary, when a variable $v \in X$ is blocked, DDS generates a D-sequent $\boldsymbol{s} \rightarrow \{v\}$ where $\boldsymbol{s}$ depends on *all* clauses that can be resolved on variable v. Such a D-sequent is, in general, much longer than $\boldsymbol{s} \rightarrow \{C\}$.

VIII. Background

The first practical QE algorithms were based on BDDs [3], [4]. Since we focus on SAT-based QE methods we do not discuss these algorithms here. The rest of QE algorithms can be roughly partitioned into two categories. The members of the first category employ various techniques to eliminate quantified variables of the formula one by one in some order [21], [1], [14], [7]. All these solvers face the same problem: there may not exist a good *single* order for variable elimination. This may lead to exponential growth of the size of intermediate formulas.

The solvers of the second category are based on enumeration of satisfying assignments [18], [12], [20], [13]. Since such assignments are, in general, "global" objects, it is hard for such solvers to follow the fine structure of the formula, *e.g.*, such solvers are not compositional [9]. That is they cannot make use of the fact that formula $\exists X_1, X_2[F_1 \wedge F_2]$ where $Vars(F_1) \cap Vars(F_2) = \emptyset$ and $X_i \subseteq Vars(F_i), i = 1, 2$ is equivalent to $\exists X_1[F_1] \wedge \exists X_2[F_2]$.

In [9], we presented a QE algorithm called DDS that employs the machinery of D-sequents based on redundancy

of variables. In a sense, DDS tries to take the best of both worlds. It branches and so can use different variable orders in different branches as the solvers of the second category. At the same time, in every branch, DDS eliminates quantified variables individually as the solvers of the first category, which makes it easier to follow the formula structure. In particular, DDS is compositional (as is $DCDS$).

Identification and removal of redundant clauses is used in preprocessing procedures of QBF-algorithms and SAT-solvers [6], [2]. Redundant clauses are also identified in the inner loop of SAT-solving (inprocessing) [19]. These procedures identify unconditional clause redundancies by recognizing the situations where such redundancies can be easily proved.

Notice that any X-clause C of a CNF formula F can be made redundant in $\exists X[F]$ as follows. Let $v \in Vars(C) \cap X$. Then $\exists X[F] \equiv \exists X[F \setminus \{C\} \cup G]$ where G is the set of all clauses obtained by resolving C with clauses of F on v. In the context of SAT-solving, this fact has been established in [11], [19]. So to make C redundant in $\exists X[F]$, one needs to add all the resolvents of C with clauses of F on a variable of $Vars(C) \cap X$. Hence, one can potentially solve QE by gradually eliminating X-clauses (including the new X-clauses produced by resolution) in some order. Unfortunately, this approach has two fundamental problems. The first problem is similar to that of variable elimination. There may not exist a *single* good order for elimination of X-clauses. The second problem is that global elimination of X-clauses one by one may lead to looping even for small formulas. That is after elimination of a number of clauses one can return to a formula seen earlier e.g. to the original formula. This is because elimination of a clause is accompanied by adding new clauses and so removed clauses may reappear again.

$DCDS$ does not have the problems above. It is not limited by one global order because in different branches of the search tree redundancy of clauses is proved in different orders. $DCDS$ does not have the problem of looping because the branches of a search tree are ordered and the algorithm cannot visit the same state twice.

IX. CONCLUSIONS

We continue to develop a calculus for solving propositional formulas with quantifiers based on the notion of dependency sequents (D-sequents). Previously, we introduced D-sequents recording redundancy of quantified *variables*. In this paper, we present a new type of D-sequents expressing redundancy of *clauses* containing quantified variables. The clause D-sequents are much more powerful that D-sequents based on variable redundancy. We use clause D-sequents to formulate a new algorithm of quantifier elimination called Derivation of Clause Dependency Sequents ($DCDS$).

We experimentally compared $DCDS$ with a QE algorithm employing D-sequents based on variable redundancy in the context of model checking. The experiments showed that $DCDS$ significantly outperformed its counterpart. We also compared a model checker based on $DCDS$ with a BDD-based model checker. We found that there was a noticeable number of benchmarks where $DCDS$ outperformed its BDD-based counterpart. These results are very promising taking into account that the current version of $DCDS$ can be drastically improved e.g. by implementing D-sequent re-using.

ACKNOWLEDGMENT

This work was funded in part by NSF grant CCF-1117184. It was also supported in part by C-FAR, one of six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA.

REFERENCES

[1] P. Abdulla, P. Bjesse, and N. Een. Symbolic reachability analysis based on sat-solvers. TACAS-00, pages 411–425, 2000.

[2] A.Biere, F.Lonsing, and M.Seidl. Blocked clause elimination for qbf. CADE-11, pages 101–115, 2011.

[3] R. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[4] P. Chauhan, E. M. Clarke, S. Jha, J.H. Kukula, H. Veith, and D. Wang. Using combinatorial optimization methods for quantification scheduling. CHARME-01, pages 293–309, 2001.

[5] E. Clarke, O. Grumberg, and D. Peled. *Model checking*. MIT Press, Cambridge, MA, USA, 1999.

[6] N. Eén and A. Biere. Effective preprocessing in sat through variable and clause elimination. In *SAT*, pages 61–75, 2005.

[7] E.Goldberg and P. Manolios. Sat-solving based on boundary point elimination. HVC-10, pages 93–111, 2010.

[8] E.Goldberg and P.Manolios. Checking satisfiability by dependency sequents. Technical Report arXiv:1207.5014 [cs.LO], 2012.

[9] E.Goldberg and P.Manolios. Quantifier elimination by dependency sequents. In *FMCAD-12*, pages 34–44, 2012.

[10] E.Goldberg and P.Manolios. Quantifier elimination by dependency sequents. Technical Report arXiv:1201.5653 [cs.LO], 2012.

[11] A. V. Gelder. Propositional search with k-clause introduction can be polynomially simulated by resolution. In *(Electronic) Proc. 5th Int'l Symposium on Artificial Intelligence and Mathematics*, 1998.

[12] H.Jin and F.Somenzi. Prime clauses for fast enumeration of satisfying assignments to boolean circuits. DAC-05, pages 750–753, 2005.

[13] J.Brauer, A.King, and J.Kriener. Existential quantification as incremental sat. CAV-11, pages 191–207, 2011.

[14] J.R.Jiang. Quantifier elimination via functional composition. In *Proceedings of the 21st International Conference on Computer Aided Verification*, CAV-09, pages 383–397, 2009.

[15] O. Kullmann. New methods for 3-sat decision and worst-case analysis. *Theor. Comput. Sci.*, 223(1-2):1–72, 1999.

[16] J. Marques-Silva and K. Sakallah. Grasp – a new search algorithm for satisfiability. In *ICCAD-96*, pages 220–227, 1996.

[17] K. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Norwell, MA, USA, 1993.

[18] K. McMillan. Applying sat methods in unbounded symbolic model checking. In *Proc. of CAV-02*, pages 250–264. Springer-Verlag, 2002.

[19] M.Järvisalo, M.Heule, and A.Biere. Inprocessing rules. IJCAR-12, pages 355–370, 2012.

[20] M.K.Ganai, A.Gupta, and P.Ashar. Efficient sat-based unbounded symbolic model checking using circuit cofactoring. ICCAD-04, pages 510–517, 2004.

[21] P. Williams, A. Biere, E. Clarke, and A. Gupta. Combining decision diagrams and sat procedures for efficient symbolic model checking. CAV-00, pages 124–138, 2000.

[22] http://fmgroup.polito.it/index.php/download/.

[23] HWMCC-2010 benchmarks, http://fmv.jku.at/hwmcc10/benchmarks.html.