

Partial Quantifier Elimination With Learning

Eugene Goldberg
email: eu.goldberg@gmail.com

Abstract—We consider a modification of the Quantifier Elimination (QE) problem called Partial QE (PQE). In PQE, only a small part of the formula is taken out of the scope of quantifiers. The appeal of PQE is that many verification problems, e.g. equivalence checking and model checking, reduce to PQE and the latter is much easier than complete QE. Earlier, we introduced a PQE algorithm based on the machinery of D-sequents. A D-sequent is a record stating that a clause is *redundant* in a quantified CNF formula in a specified subspace. To make this algorithm efficient, it is important to reuse learned D-sequents. However, reusing D-sequents is not as easy as conflict clauses in SAT-solvers because redundancy is a *structural* rather than a semantic property. In [21], we modified the definition of D-sequents to enable their *safe* reusing. In this paper, we present a PQE algorithm based on new D-sequents. It is different from its predecessor in two aspects. First, the new algorithm can learn and reuse D-sequents. Second, it proves clauses redundant one by one and thus backtracks as soon as the current target clause is proved redundant in the current subspace. This makes the new PQE algorithm similar to a SAT-solver that backtracks as soon as just *one* clause is falsified. We show experimentally that the new PQE algorithm outperforms its predecessor.

I. INTRODUCTION

Many verification problems reduce to Quantifier Elimination (QE). So, any progress in QE is of great importance. In this paper, we consider propositional CNF formulas with existential quantifiers. Given formula $\exists X[F(X, Y)]$ where X and Y are sets of variables, the QE problem is to find a quantifier-free formula $F^*(Y)$ such that $F^* \equiv \exists X[F]$. Building a practical QE algorithm is a tall order. In addition to the sheer complexity of QE, a major obstacle here is that the size of formula $F^*(Y)$ can be prohibitively large.

There are at least two ways of making QE easier to solve. First, one can consider only instances of QE where $|Y|$ is small, which limits the size of F^* . In particular, if $|Y| = 0$, QE reduces to the satisfiability problem (SAT). This line of research featuring very efficient methods of model checking based on SAT [3], [35], [7] has gained great popularity. Another way to address the complexity of QE suggested in [25] is to perform **partial QE (PQE)**. Given formula $\exists X[F_1(X, Y) \wedge F_2(X, Y)]$, the PQE problem is to find a quantifier-free formula $F_1^*(Y)$ such that $F_1^* \wedge \exists X[F_2] \equiv \exists X[F_1 \wedge F_2]$. We will say that formula F_1^* is obtained by *taking F_1 out of the scope of quantifiers*.

The appeal of PQE is threefold. First, intuitively, PQE should be much simpler than QE if F_1 is much smaller than F_2 . Second, PQE can perform SAT. So one can view a PQE-solver as a SAT-solver with extra semantic power due to using quantifiers. Third, in addition to SAT, many verification problems, reduce to PQE (see Section III). For instance, an

equivalence checker based on PQE [19] enables construction of short resolution proofs of equivalence for a very broad class of structurally similar circuits. These proofs are based on the notion of clause redundancy¹ in a *quantified* formula and thus cannot be generated by a traditional SAT-solver. In [20], we show that a PQE-solver can check if the reachability diameter exceeds a specified value. So it can turn bounded model checking [3] into unbounded as opposed to a pure SAT-solver. Importantly, no generation of an *inductive invariant* is required by the method of [20].

If $F_1^*(Y)$ is a solution to the PQE problem above, it is implied by $F_1 \wedge F_2$. So F_1^* can be obtained by resolving clauses of $F_1 \wedge F_2$. However, a PQE-solver based on resolution *alone* cannot efficiently address the following “termination problem”. Suppose one builds F_1^* incrementally, adding one clause at a time. When can one terminate this procedure claiming that F^* is a solution to the PQE problem (and so $F_1^* \wedge \exists X[F_2] \equiv \exists X[F_1 \wedge F_2]$)? The inability of resolution to address the termination problem stems from its “asymmetry” in treating satisfiable and unsatisfiable formulas. To prove formula G unsatisfiable, one just needs to add to G new resolvents until an empty clause is derived. However, proving G *satisfiable* requires reaching a saturation point where every new resolvent is implied by a clause of G .

In [22], [24] we approached the termination problem above using the following observation. Assume for the sake of simplicity that every clause of F_1 contains at least one variable of X . Then, if F_1^* is a solution, F_1 can be dropped from $F_1^* \wedge \exists X[F_1 \wedge F_2]$. Thus, F_1^* becomes a solution as soon as it makes the clauses of F_1 *redundant*. The ability of redundancy-based reasoning to handle the termination problem is rooted in the fact that such reasoning enables treating satisfiable and unsatisfiable formulas in a symmetric way (see Section V).

In [25], we introduced a PQE-solver called *DS-PQE* based on the notion of redundancy (DS stands for “D-Sequent”). *DS-PQE* is a branching algorithm that, in addition to deriving new clauses and conjoining them with $F_1 \wedge F_2$, generates dependency sequents (*D-sequents*). A D-sequent is a record saying that a clause is redundant in a specified subspace. *DS-PQE* branches until proving redundancy² of target clauses becomes trivial at which point so-called “atomic” D-sequents are generated. The D-sequents of different branches are merged using a resolution-like operation called *join*. Upon

¹A clause is a disjunction of literals. So a CNF formula F is a conjunction of clauses: $C_1 \wedge \dots \wedge C_k$. We also consider F as the *set* of clauses $\{C_1, \dots, C_k\}$. Clause C is redundant in $\exists X[F]$ if $\exists X[F] \equiv \exists X[F \setminus \{C\}]$.

²By “proving a clause C redundant” we mean “making C redundant by adding new clauses (if necessary) and then proving C redundant”.

completing the search tree, *DS-PQE* derives D-sequents stating redundancy of the clauses of F_1 .

DS-PQE has two flaws. First, *DS-PQE* employs “multi-event” backtracking. Namely, it backtracks only when *all* clauses of F_1 are proved redundant in the current subspace. (This is different from a SAT-solver that backtracks as soon as *just one* clause of the formula is falsified.) The intuition here is that multi-event backtracking leads to building very deep and thus very large search trees. Second, *DS-PQE* does not reuse D-sequents derived in different branches. The problem here is that redundancy is a *structural* rather than a *semantic* property. So, a clause redundant in formula G' may not be redundant in G'' logically equivalent to G' (whereas a semantic property holds for *all* equivalent formulas). So, reusing a D-sequent is not as easy as reusing a clause learned by a SAT-solver.

In this paper, we address both flaws of *DS-PQE*. First, we present a new PQE algorithm called *DS-PQE⁺* that employs *single-event backtracking*. At any given moment, *DS-PQE⁺* proves redundancy of only one clause. Once this goal is achieved, it picks a new clause to prove redundant. Second, *DS-PQE⁺* uses a new definition of D-sequents introduced in [21]. This definition facilitates safe reusing of D-sequents. We show experimentally that *DS-PQE⁺* is significantly faster than *DS-PQE*.

This main body of the paper³ is structured as follows. Basic definitions are given in Section II. Section III lists some verification problems that reduce to PQE. In Section IV, we give a simple example of using the machinery of D-sequents in PQE. Section V explains how to use the notion of redundancy to treat satisfiable and unsatisfiable formulas “symmetrically”. In Section VI, we describe the semantics of clause redundancy in terms of boundary points. Sections VII-X present the machinery of D-sequents. Section XIII provides experimental results. Some background is given in Section XIV. Finally, in Sections XV and XVI, we make conclusions and describe directions for future research.

II. BASIC DEFINITIONS

In this paper, we consider only propositional CNF formulas. In the sequel, when we say “formula” without mentioning quantifiers we mean a *quantifier-free CNF* formula.

Definition 1: Let F be a CNF formula and X be a subset of variables of F . We will refer to $\exists X[F]$ as an **\exists CNF formula**.

Definition 2: Let F be a CNF formula. $\mathbf{Vars}(F)$ denotes the set of variables of F and $\mathbf{Vars}(\exists X[F])$ denotes $\mathbf{Vars}(F) \setminus X$.

Definition 3: Let V be a set of variables. An **assignment** \vec{q} to V is a mapping $V' \rightarrow \{0, 1\}$ where $V' \subseteq V$. We will denote the set of variables assigned in \vec{q} as $\mathbf{Vars}(\vec{q})$. We will denote as $\vec{q} \subseteq \vec{r}$ the fact that a) $\mathbf{Vars}(\vec{q}) \subseteq \mathbf{Vars}(\vec{r})$ and b) every variable of $\mathbf{Vars}(\vec{q})$ has the same value in \vec{q} and \vec{r} .

Definition 4: Let C be a clause, H be a formula that may have quantifiers, and \vec{q} be an assignment. $C_{\vec{q}} \equiv 1$ if C is satisfied by \vec{q} ; otherwise it is the clause obtained from C by

removing all literals falsified by \vec{q} . $H_{\vec{q}}$ denotes the formula obtained from H by replacing every clause C with $C_{\vec{q}}$.

Definition 5: Let G, H be formulas that may have quantifiers. We say that G, H are **equivalent**, written $G \equiv H$, if for all assignments \vec{q} where $\mathbf{Vars}(\vec{q}) \supseteq (\mathbf{Vars}(G) \cup \mathbf{Vars}(H))$, we have $G_{\vec{q}} = H_{\vec{q}}$.

Definition 6: The **Quantifier Elimination (QE)** problem for formula $\exists X[F(X, Y)]$ is to find a formula $F^*(Y)$ such that $F^* \equiv \exists X[F]$.

Definition 7: The **Partial QE (PQE)** problem of taking F_1 out of the scope of quantifiers in $\exists X[F_1(X, Y) \wedge F_2(X, Y)]$ is to find formula $F_1^*(Y)$ such that $F_1^* \wedge \exists X[F_2] \equiv \exists X[F_1 \wedge F_2]$.

Remark 1: From now on, we will use X and Y to denote sets of quantified and non-quantified variables respectively. We will assume that variables denoted by x_i and y_i are in X and Y respectively. Using X, Y in a quantifier-free formula implies that in the context of QE/PQE, X and Y specify the quantified and non-quantified variables respectively.

Definition 8: Let $\exists X[F(X, Y)]$ be an \exists CNF formula. A clause C of F is called an **X-clause** if $\mathbf{Vars}(C) \cap X \neq \emptyset$.

Definition 9: Let F be a CNF formula and $G \subseteq F$ and $G \neq \emptyset$. The clauses of G are **redundant in F** if $F \equiv (F \setminus G)$. The clauses of G are **redundant in $\exists X[F]$** if $\exists X[F] \equiv \exists X[F \setminus G]$. Note that $F \equiv (F \setminus G)$ implies $\exists X[F] \equiv \exists X[F \setminus G]$ but the opposite is not true.

III. SOME APPLICATIONS OF PQE

In this section, we justify our interest in PQE by listing some verification problems that can be solved by a PQE algorithm.

A. Circuit-SAT

Let $N(X, Y, Z)$ be a combinational circuit where X, Y, Z are sets of input, internal and output variables respectively. Let \vec{z} be an assignment⁴ to Z . Consider the problem⁵ of finding inputs (i.e. assignments to X) for which N produces output \vec{z} . Let $F(X, Y, Z)$ be a formula specifying N . Let C_z denote the longest clause falsified by \vec{z} . The problem above reduces to taking C_z out of the scope of quantifiers in $\exists W[C_z \wedge F]$ where $W = Y \cup Z$ (see [26]). Namely, one needs to find a formula $G(X)$ such that $\exists W[C_z \wedge F] \equiv G \wedge \exists W[F]$. Every input \vec{x} falsifying G produces the output \vec{z} .

B. General SAT

Let $F(X)$ be a formula to be checked for satisfiability and \vec{x} be an assignment to X . Let F_1 and F_2 denote the clauses of F satisfied and falsified by \vec{x} respectively. Then checking the satisfiability of F reduces to taking F_1 out of the scope of quantifiers in $\exists X[F_1 \wedge F_2]$ (see [25]). That is one just needs to find F_1^* such that $\exists X[F_1 \wedge F_2] \equiv F_1^* \wedge \exists X[F_2]$. Since all variables of F are quantified, F_1^* is a constant. If $F_1^* = \text{false}$, F is unsatisfiable because $\exists X[F] \equiv \exists X[F_1 \wedge F_2] \equiv \text{false}$. If $F_1^* = \text{true}$, F is satisfiable (because F_2 is satisfied by \vec{x}).

⁴In this section, when we say “an assignment \vec{v} to a set of variables V ” we mean a full assignment (i.e. every variable of V is assigned in \vec{v}).

⁵This problem reduces to Circuit-SAT if $Z = \{z\}$, $\vec{z} = (z = 1)$ and it suffices to produce just one input (if any) for which N outputs \vec{z} .

³Some additional information is provided in Appendices.

C. Interpolation

Let $I(Y)$ be an interpolant for formulas $A(X, Y)$ and $\overline{B}(Y, Z)$ i.e. $A \Rightarrow I \Rightarrow \overline{B}$. Let formula $A^*(Y)$ be obtained by taking A out of the scope of quantifiers in $\exists W[A \wedge B]$ where $W = X \cup Z$. That is $\exists W[A \wedge B] \equiv A^* \wedge \exists W[B]$. Assume also that $A \Rightarrow A^*$. Then $A \Rightarrow A^* \Rightarrow \overline{B}$ and A^* is an interpolant [18]. So, one can view interpolation as a special case of PQE.

D. Equivalence checking

Let $N'(X', Y', z')$ and $N''(X'', Y'', z'')$ be single-output combinational circuits to be checked for equivalence. Here X', Y' are sets of input and internal variables and z' is the output variable of N' . Sets X'' and Y'' and variable z'' have the same meaning for N'' . Let $EQ(X', X'')$ specify the predicate such that $EQ(\vec{x}', \vec{x}'')$ iff $\vec{x}' = \vec{x}''$. Let formulas $F'(X', Y', z')$ and $F''(X'', Y'', z'')$ specify circuits N' and N'' respectively.

The equivalence of N' and N'' can be checked by taking EQ from the scope of quantifiers in $\exists W[EQ \wedge F' \wedge F'']$ where $W = X' \cup X'' \cup Y' \cup Y''$ (see [19]). Let $h(z', z'')$ be a formula such that $\exists W[EQ \wedge F' \wedge F''] \equiv h \wedge \exists W[F' \wedge F'']$. If $h(z', z'')$ specifies $z' \equiv z''$, then N' and N'' are equivalent. Otherwise, N' and N'' are inequivalent unless they implement identical constants. (This possibility can be ruled out by a few easy SAT-checks.)

E. Model checking

Let formulas $T(S, S')$ and $I(S)$ specify the transition relation and initial states of a system ξ respectively. Here S and S' are sets of variables specifying the present and next states respectively. Let $Diam(I, T)$ denote the *reachability diameter* of ξ (i.e. every state of ξ is reachable in at most $Diam(I, T)$ transitions).

Given a number n , one can use a PQE solver to check if $n \geq Diam(I, T)$ as follows [20]. Let $I_1 = I(S_1)$ and $W_n = S_0 \cup \dots \cup S_n$ and $G_{0,n} = T_{0,1} \wedge \dots \wedge T_{n-1,n}$ and $T_{i,i+1} = T(S_i, S_{i+1})$. Testing if $n \geq Diam(I, T)$ reduces to checking if I_1 is redundant in $\exists W_n[I_1 \wedge I_0 \wedge G_{0,n+1}]$ i.e. whether $\exists W_n[I_1 \wedge I_0 \wedge G_{0,n+1}] \equiv \exists W_n[I_0 \wedge G_{0,n+1}]$. If so, then $n \geq Diam(I, T)$. Then, to prove a safety property $P(S)$, it suffices to run BMC [3] to show that no counterexample of length n or less exists.

IV. A SIMPLE EXAMPLE

In this section, we present a simple example of performing PQE by deriving D-sequents. A D-sequent of [24] is a record $(\exists X[F], \vec{q}) \rightarrow C$ stating redundancy of clause C in $\exists X[F]$ in subspace \vec{q} (where \vec{q} is an assignment to variables of F). Let $\exists X[C_1 \wedge G]$ be a formula where $X = \{x_1, x_2\}$, $C_1 = \overline{x}_1 \vee x_2$, $G = C_2 \wedge C_3$, $C_2 = y \vee x_1$, $C_3 = y \vee \overline{x}_2$. Consider the PQE problem of taking C_1 out of the scope of quantifiers. Below we solve this problem by proving C_1 redundant.

In subspace $y=0$, clauses C_2, C_3 are **unit** (i.e. one literal is unassigned, the rest are falsified). After assigning $x_1=1$, $x_2=0$ to satisfy C_2, C_3 , the clause C_1 is falsified. Using the

standard conflict analysis [33] one derives a conflict clause $C_4 = y$. Adding C_4 to $C_1 \wedge G$ makes C_1 redundant in subspace $y=0$. So the D-sequent S' equal to $(\exists X[F], \vec{q}') \rightarrow C_1$ holds where $F = C_1 \wedge G \wedge C_4$ and $\vec{q}' = (y=0)$.

In subspace $y=1$, the clause C_1 is “blocked” at x_1 . That is no clause of F is resolvable with C_1 on x_1 in subspace $y=1$ because C_2 is satisfied by $y=1$ (see Subsection IX-C). So C_1 is redundant in formula $\exists X[F]$ and the D-sequent S'' equal to $(\exists X[F], \vec{q}'') \rightarrow C_1$ holds where $\vec{q}'' = (y=1)$. D-sequents S' and S'' are examples of so-called atomic D-sequents. They are derived when proving clause redundancy is trivial (see Section IX). One can produce a new D-sequent $(\exists X[F], \vec{q}) \rightarrow C_1$ where $\vec{q} = \emptyset$ by “joining” S' and S'' at y (see Subsection X-A). This D-sequent states the *unconditional* redundancy of C_1 in $\exists X[F]$. So, $C_4 \wedge \exists X[G] \equiv \exists X[C_1 \wedge G \wedge C_4]$. Since $C_1 \wedge G$ implies C_4 , then $C_4 \wedge \exists X[G] \equiv \exists X[C_1 \wedge G]$. So C_4 is a solution to our PQE problem.

V. REDUNDANCY AND SAT/UNSAT SYMMETRY

As mentioned earlier, there is an obvious asymmetry in how pure resolution treats satisfiable and unsatisfiable formulas. Namely, resolution cannot efficiently solve satisfiable formulas. In the SAT-solvers based on the DPLL procedure [13], this problem is addressed by building a search tree where each branch corresponds to an assignment. For a satisfiable formula, the search terminates as soon as a branch specifying a satisfying assignment is found.

Note that the DPLL procedure does not eliminate the asymmetry in treating satisfiable and unsatisfiable formulas. It just simplifies proving a formula satisfiable (by finding a satisfying assignment). However, this does not work well when one has to enumerate *many* satisfying assignments. Consider, for instance, the QE problem of finding $F^*(Y)$ logically equivalent to $\exists X[F(X, Y)]$. Suppose one builds $F^*(Y)$ by a DPLL-like procedure. In the worst case, this requires finding a satisfying assignment for every assignment \vec{y} to Y for which F is satisfiable.

One can use the notion of redundancy to recover the symmetry between satisfiable and unsatisfiable formulas. Consider, for instance, the QE problem above. Let F be unsatisfiable in subspace \vec{y} . Then to make the X -clauses of F redundant in this subspace one needs to add a clause $C(Y)$ (implied by F) that is falsified by \vec{y} . If F is satisfiable in subspace \vec{y} , all X -clauses are already redundant in this subspace and hence no clause needs to be added. So, the only difference between SAT and UNSAT cases is that in the UNSAT case one has to add a clause to make target X -clauses redundant.

VI. PROVING CLAUSE REDUNDANCY

Let $F(X)$ be a quantifier-free formula. Proving redundancy of a clause $C \in F$ reduces to checking if $F \setminus \{C\}$ implies C . So, in this case, a redundancy check is straightforward and reduces to SAT. Now consider proving redundancy of $C \in F$ in formula $\exists X[F(X, Y)]$. If $Vars(C) \subseteq Y$, then the redundancy check is still the same as above. However, the situation changes if C is an X -clause. The fact that an

X -clause is redundant in $\exists X[F]$ does not mean that it is redundant in F as well.

In [23], to address the problem of proving clause redundancy, we developed a machinery of boundary points. Given a formula $\exists X[F]$ and a clause $C \in F$, a boundary point is a full assignment (\vec{x}, \vec{y}) to $X \cup Y$ that falsifies C but satisfies $F \setminus \{C\}$. This boundary point is called removable if there is a clause $B(Y)$ implied by F that is falsified by (\vec{x}, \vec{y}) . Adding B to F eliminates (\vec{x}, \vec{y}) as a boundary point (because it does not satisfy $F \setminus \{C\}$ anymore). An X -clause C is redundant in $\exists X[F]$ if no removable boundary point exists.

VII. DEPENDENCY SEQUENTS (D-SEQUENTS)

In [24], we introduced a machinery of D-sequents meant for dealing with quantified formulas. It can be viewed as an extension of resolution that facilitates treating satisfiable and unsatisfiable formulas in a symmetric way (see Section V). In this section, we modify the definition of D-sequents introduced in [24]. In Subsection VII-A, we explain the reason for such a modification. The new definition is given in Subsection VII-B.

A. Motivating example

Let formula $\exists X[F]$ contain two identical X -clauses C and B . The presence of C makes B redundant and vice versa. So, D-sequents $(\exists X[F], \vec{q}) \rightarrow C$ and $(\exists X[F], \vec{q}) \rightarrow B$ hold where $\vec{q} = \emptyset$. Denote them as S_C and S_B respectively. (Here, we use the old definition of D-sequents given in [24].) S_C and S_B state that C and B are redundant in $\exists X[F]$ *individually*. Using S_C and S_B *together* (to remove *both* B and C from $\exists X[F]$) is incorrect because it involves circular reasoning.

The problem here is that redundancy is a *structural* property. So, the redundancy of B in $\exists X[F]$ does not imply that of B in $\exists X[F \setminus \{C\}]$ even though $F \equiv F \setminus \{C\}$. The definition of a D-sequent given in [24] does not help to address the problem above. This definition states redundancy of a clause only with respect to formula $\exists X[F]$. (This makes it hard to reuse D-sequents and is the reason why the PQE-solver introduced in [24] does not reuse D-sequents). We address this problem by adding a *structural constraint* to the definition of a D-sequent. It specifies a *subset of formulas* where a D-sequent holds and so helps to avoid using this D-sequent in situations where it *may not hold*. Adding structural constraints to D-sequents S_C and S_B makes them mutually exclusive (see Example 1 below).

B. Definition of D-sequents

Definition 10: Let $\exists X[F]$ be an \exists CNF formula and \vec{q} be an assignment to $\text{Vars}(F)$. Let C be an X -clause of F and H be a subset of $F \setminus \{C\}$. A dependency sequent (**D-sequent**) S has the form $(\exists X[F], \vec{q}, H) \rightarrow C$. It states that clause $C_{\vec{q}}$ is redundant in every formula $\exists X[W_{\vec{q}}]$ logically equivalent to $\exists X[F_{\vec{q}}]$ where $H \cup \{C\} \subseteq W \subseteq F$.

Definition 11: The assignment \vec{q} and formula H above are called the **conditional** and the **structure constraint** of the D-sequent S respectively. We will call $\exists X[W]$, where $H \cup \{C\} \subseteq W \subseteq F$, a **member formula** of S . We will say

that a D-sequent S specified by $(\exists X[F], \vec{q}, H) \rightarrow C$ **holds** if it states redundancy of C according to Definition 10 (i.e. if S is *correct*). We will say that S is **applicable** to a formula $\exists X[W]$ if the latter is a member formula of S . Otherwise, S is called inapplicable to $\exists X[W]$.

The structure constraint H of Definition 10 specifies a subset of formulas logically equivalent to $\exists X[F]$ where the clause C is redundant. From a practical point of view, the presence of H influences the order in which X -clauses can be proved redundant. Proving an X -clause B of H redundant and removing it from F renders the D-sequent S inapplicable to the modified formula (i.e. $\exists X[F \setminus \{B\}]$). Thus, if one intends to use S , the clause B should be proved redundant *after* C .

Example 1: Consider the example introduced in Subsection VII-A. In terms of Definition 10, the D-sequent S_C looks like $(\exists X[F], \vec{q}, H_C) \rightarrow C$ where $\vec{q} = \emptyset$, $H_C = \{B\}$ (because the presence of clause B is used to prove C redundant). Similarly, the D-sequent S_B looks like $(\exists X[F], \vec{q}, H_B) \rightarrow B$ where $H_B = \{C\}$. D-sequents S_C and S_B are mutually exclusive: using S_C to remove C from F as a redundant clause renders S_B inapplicable and vice versa.

Remark 2: We will abbreviate D-sequent $(\exists X[F], \vec{q}, H) \rightarrow C$ to $(\vec{q}, H) \rightarrow C$ if $\exists X[F]$ is known from the context.

VIII. REUSING SINGLE AND MULTIPLE D-SEQUENTS

In this section, we discuss conditions under which single and multiple D-sequents can be safely reused.

A. Reusing a single D-sequent

Let S be a D-sequent specified by $(\exists X[F], \vec{q}, H) \rightarrow C$. We will say that S is **active** in subspace \vec{r} for formula $\exists X[W]$ if

- $\vec{q} \subseteq \vec{r}$ and
- S is applicable to $\exists X[W]$ (see Definition 11)

The activation of S means that it can be safely *reused* (i.e. C can be dropped in the subspace \vec{r} as redundant⁶ in $\exists X[W]$).

An applicable D-sequent S equal to $(\exists X[F], \vec{q}, H) \rightarrow C$ is called **unit** under assignment \vec{r} if all values assigned in \vec{q} *but one* are present in \vec{r} . Suppose, for instance, $\vec{q} = (y_1 = 0, x_5 = 1)$ and \vec{r} contains $y_1 = 0$ but x_5 is not assigned in \vec{r} . Then S is unit. Adding the assignment $x_5 = 1$ to \vec{r} , activates S , which indicates that C is redundant in the subspace $\vec{r} \cup \{x_5 = 1\}$. So, a unit D-sequent can be used like a unit clause in Boolean Constraint Propagation (BCP) of a SAT-solver. Namely, one can use S to derive the “deactivating” assignment $x_5 = 0$ as a direction to a subspace where C is not proved redundant yet.

B. Reusing a set of D-sequents

In Example 1, we described D-sequents that *cannot* be active together. Below, we introduce a condition under which a set of D-sequents *can* be active together.

Definition 12: Assignments \vec{q}' and \vec{q}'' are called **compatible** if every variable of $\text{Vars}(\vec{q}') \cap \text{Vars}(\vec{q}'')$ is assigned the same value in \vec{q}' and \vec{q}'' .

⁶Redundancy of a clause in subspace \vec{q} does not *trivially* imply its redundancy in subspace $\vec{q} \subset \vec{r}$, i.e. in a smaller subspace (see Appendix I).

Definition 13: Let $\exists X[F]$ be an \exists CNF formula. Let S_1, \dots, S_k be D-sequents specified by $(\vec{q}_1, H_1) \rightarrow C_1, \dots, (\vec{q}_k, H_k) \rightarrow C_k$ respectively. They are called **consistent** if a) every pair $\vec{q}_i, \vec{q}_j, 1 \leq i, j \leq k$ is compatible and b) there is an order π on $\{1, \dots, k\}$ such that $\exists X[F \setminus \{C_{\pi(1)}, \dots, C_{\pi(m-1)}\}]$ obtained after using D-sequents $S_{\pi(1)}, \dots, S_{\pi(m-1)}$ is a member formula of $S_{\pi(m)}, \forall m \in \{2, \dots, k\}$.

The item b) above means that S_1, \dots, S_k can be active together if there is an order π following which one guarantees the applicability of *every D-sequent*. (The D-sequents S_C and S_B of Example 1 are *inconsistent* because such an order does not exist. Applying one D-sequent makes the other inapplicable.) Definition 13 specifies a *sufficient* condition for a set of D-sequents to be active together in a subspace \vec{r} where $\vec{q}_i \subseteq \vec{r}, 1 \leq i \leq k$. If this condition is met, C_1, \dots, C_k can be safely removed from $\exists X[F]$ in the subspace \vec{r} (see [21]).

IX. ATOMIC D-SEQUENTS

In this section, we describe D-sequents called atomic. An atomic D-sequent is generated when proving a clause redundant is trivial [24]. We modify the definitions of [24] to accommodate the appearance of a structure constraint.

A. Atomic D-sequents of the first kind

Proposition 1: Let $\exists X[F]$ be an \exists CNF formula and $C \in F$ and $v \in \text{Vars}(C)$. Let $v = b$ where $b \in \{0, 1\}$ satisfy C . Then the D-sequent $(\vec{q}, H) \rightarrow C$ holds where $\vec{q} = (v = b)$ and $H = \emptyset$. We will refer to it as an atomic D-sequent of the **first kind**.

Proofs of all propositions can be found in [21]. Satisfying C by an assignment does not require the presence of any other clause of F . Hence, the structure constraint of a D-sequent of the first kind is an empty set of clauses.

Example 2: Let $\exists X[F]$ be an \exists CNF formula and $C = y_1 \vee \bar{x}_5$ be a clause of F . Since C is satisfied by assignments $y_1 = 1$ and $x_5 = 0$, D-sequents $(y_1 = 1, \emptyset) \rightarrow C$ and $(x_5 = 0, \emptyset) \rightarrow C$ hold.

B. Atomic D-sequents of the second kind

Proposition 2: Let $\exists X[F]$ be an \exists CNF formula and \vec{q} be an assignment to $\text{Vars}(F)$. Let C and B be clauses of F and C be an X -clause. Let $C_{\vec{q}}$ still be an X -clause and $B_{\vec{q}}$ imply $C_{\vec{q}}$ (i.e. every literal of $B_{\vec{q}}$ is in $C_{\vec{q}}$). Then the D-sequent $(\vec{q}, H) \rightarrow C$ holds where $H = \{B\}$. We will refer to it as an atomic D-sequent of the **second kind**.

Example 3: Let $\exists X[F]$ be an \exists CNF formula. Let $B = y_1 \vee x_2$ and $C = x_2 \vee \bar{x}_3$ be clauses of F . Let $\vec{q} = (y_1 = 0)$. Since $B_{\vec{q}}$ implies $C_{\vec{q}}$ the D-sequent $(\vec{q}, \{B\}) \rightarrow C$ holds.

C. Atomic D-sequents of the third kind

Definition 14: Let clauses C', C'' have opposite literals of exactly one variable $v \in \text{Vars}(C') \cap \text{Vars}(C'')$. The clause C having all literals of C', C'' but those of v is called the **resolvent** of C', C'' on v . The clause C is said to be obtained by **resolution** on v . Clauses C', C'' are called **resolvable** on v .

Definition 15: 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 [31].

If a clause C of an \exists CNF formula is blocked with respect to a quantified variable in a subspace, it is redundant in this subspace. This fact is used by the proposition below.

Proposition 3: Let $\exists X[F]$ be an \exists CNF formula. Let C be an X -clause of F and $v \in (\text{Vars}(C) \cap X)$. Let C_1, \dots, C_k be the clauses of F resolvable with C on variable v . Let $(\vec{q}_1, H_1) \rightarrow C_1, \dots, (\vec{q}_k, H_k) \rightarrow C_k$ be consistent D-sequents (see Definition 13). Then the D-sequent $(q, H) \rightarrow C$ holds where $\vec{q} = \bigcup_{i=1}^{i=k} \vec{q}_i$ and $H = \bigcup_{i=1}^{i=k} H_i$. We will refer to it as an atomic D-sequent of the **third kind**.

Example 4: Let $\exists X[F]$ be an \exists CNF formula. Let C_1, C_2, C_3 be the only clauses of F with variable $x_1 \in X$ where $C_1 = x_1 \vee x_2, C_2 = y_1 \vee \bar{x}_1, C_3 = y_2 \vee \bar{x}_1$. Since $y_1 = 1$ satisfies C_2 , the D-sequent $(y_1 = 1, \emptyset) \rightarrow C_2$ holds. Suppose that the D-sequent $(x_2 = 1, \{C_4\}) \rightarrow C_3$ holds where $C_4 \in F$. Note that the two D-sequents above are consistent. So, from Proposition 3 it follows that the D-sequent $(\vec{q}, \{C_4\}) \rightarrow C_1$ holds where $\vec{q} = (y_1 = 1, x_2 = 1)$. The clause C_1 is redundant in the subspace \vec{q} because it is blocked at x_1 in this subspace.

X. JOINING AND UPDATING D-SEQUENTS

In this section, we recall two methods for producing a new D-sequent from existing ones [24]. In Subsection X-A, we present a resolution-like operation called *join* that produces a new D-sequent from two parent D-sequents. Subsection X-B describes how a D-sequent is recomputed after adding implications. We modify the description of these methods given in [24] to accommodate the appearance of a structure constraint. In Appendix II, we describe one more way to produce a new D-sequent that was not described in [24].

A. Join operation

Definition 16: Let \vec{q}' and \vec{q}'' be assignments in which exactly one variable $v \in \text{Vars}(\vec{q}') \cap \text{Vars}(\vec{q}'')$ is assigned different values. The assignment \vec{q} consisting of all the assignments of \vec{q}' and \vec{q}'' but those to v is called the *resolvent* of \vec{q}', \vec{q}'' on v . Assignments \vec{q}', \vec{q}'' are called *resolvable* on v .

Proposition 4: Let $\exists X[F]$ be an \exists CNF formula. Let D-sequents $(\vec{q}', H') \rightarrow C$ and $(\vec{q}'', H'') \rightarrow C$ hold. Let \vec{q}, \vec{q}'' be resolvable on v and \vec{q} be the resolvent. Then the D-sequent $(\vec{q}, H) \rightarrow C$ holds where $H = H' \cup H''$.

Definition 17: We will say that the D-sequent $(\vec{q}, H) \rightarrow C$ of Proposition 4 is produced by **joining D-sequents** $(\vec{q}', H') \rightarrow C$ and $(\vec{q}'', H'') \rightarrow C$ **at variable** v .

Example 5: Let $\exists X[F(X, Y)]$ be an \exists CNF formula. Let C_1, C_2, C_3 be clauses of F and C_1 be an X -clause. Let $(\vec{q}', H') \rightarrow C_1, (\vec{q}'', H'') \rightarrow C_1$ be D-sequents where $\vec{q}' = (y_1 = 0, x_1 = 0), \vec{q}'' = (y_1 = 1, x_2 = 1), H' = \{C_2\}, H'' = \{C_3\}$. By joining them at y_1 , one produces the D-sequent $(\vec{q}, H) \rightarrow C_1$ where $\vec{q} = (x_1 = 0, x_2 = 1)$ and $H = \{C_2, C_3\}$.

B. Updating D-sequents after adding an implication

As we mentioned earlier, proving redundancy of X -clauses of $\exists X[F]$ requires adding new clauses implied by F . The

proposition below shows that the D-sequents learned for $\exists X[F]$ before can be trivially updated.

Proposition 5: Let D-sequent $(\exists X[F], \vec{q}, H) \rightarrow C$ hold and R be a formula implied by F . Then the D-sequent $(\exists X[F \wedge R], \vec{q}, H) \rightarrow C$ holds too.

XI. INTRODUCING $DS-PQE^+$

In this section, we describe a PQE-algorithm called $DS-PQE^+$. As we mentioned earlier, in contrast to $DS-PQE$ of [25], $DS-PQE^+$ uses single-event backtracking. Namely, $DS-PQE^+$ proves redundancy of X -clauses one by one and backtracks as soon as the current target X -clause is proved redundant in the current subspace. Besides, due to introduction of structure constraints, it is safe for $DS-PQE^+$ to reuse D-sequents. A proof of correctness of $DS-PQE^+$ is given in Appendix VI.

A. Main loop of $DS-PQE^+$

```

 $DS-PQE^+(F_1, F_2 \parallel X) \{$ 
1  $Ds := \emptyset$ 
2 while ( $true$ )  $\{$ 
3  $C := PickXcls(F_1)$ 
4 if ( $C = nil$ ) return( $F_1$ )
5  $PrvRed(F_1, F_2, Ds \parallel C, X)$ 
6  $F_1 := F_1 \setminus \{C\}$ 

```

Fig. 1. Main loop

The main loop of $DS-PQE^+$ is shown in Fig. 1. $DS-PQE^+$ accepts formulas $F_1(X, Y), F_2(X, Y)$ and set X and outputs formula $F_1^*(Y)$ such that $\exists X[F_1 \wedge F_2] \equiv F_1^* \wedge \exists X[F_2]$. We use symbol ‘ \parallel ’ to separate in/out-parameters and in-parameters. For instance, the line $DS-PQE^+(F_1, F_2 \parallel X)$ means that formulas F_1, F_2 change by $DS-PQE^+$ (via adding/removing clauses) whereas X does not.

$DS-PQE^+$ first initializes the set Ds of learned D-sequents. It starts an iteration of the loop with picking an X -clause $C \in F_1$ (line 3). If every clause of F_1 contains only variables of Y , then F_1 is a solution $F_1^*(Y)$ to the PQE problem above (line 4). Otherwise, $DS-PQE^+$ invokes a procedure called $PrvRed$ to prove C redundant. This may require adding new clauses to F_1 and F_2 . In particular, $PrvRed$ may add to F_1 new X -clauses to be proved redundant in later iterations of the loop. Finally, $DS-PQE^+$ removes C from F_1 (line 6).

B. Description of $PrvRed$ procedure

The pseudo-code of $PrvRed$ is shown in Fig 2. The objective of $PrvRed$ is to prove a clause of F_1 redundant. We will refer to this clause as the *primary target* and denote it as C_{pr} . To prove C_{pr} redundant, $PrvRed$, in general, needs to prove redundancy of other X -clauses called *secondary targets*. At any given moment, $PrvRed$ tries to prove redundancy of only one X -clause. If a new secondary target is selected, the current target is pushed on a stack T to be finished later. How $DS-PQE^+$ manages secondary targets is described in Section XII. (The lines of code relevant to this part of $DS-PQE^+$ are marked in Fig. 2 and 3 with an asterisk.)

First, $PrvRed$ initializes its variables (lines 1-3). The stack T of the target X -clauses is initialized to C_{pr} . The current assignment \vec{a} to $X \cup Y$ is initially empty. So is the assignment queue Q . The current target clause C_{trg} is set to C_{pr} . The main

work is done in a while loop that is similar to the main loop of a SAT-solver [33]. In particular, $PrvRed$ uses the notion of a *decision level*. The latter consists of a decision assignment and implied assignments derived by BCP. Decision level number 0 is an exception: it consists only of implied assignments. BCP derives implied assignments from unit clauses and from unit D-sequents (see Subsection VIII-A).

```

//  $\eta$  denotes  $(Q, \vec{a}, T, C_{trg})$ 
//  $\xi$  denotes  $(F_1, F_2, Ds, X)$ 
//  $\phi$  denotes  $(F_1, F_2, Ds, \vec{a}, T)$ 
//
 $PrvRed(F_1, F_2, Ds \parallel C_{pr}, X) \{$ 
1  $T := InitStack(C_{pr})$ 
2  $\vec{a} := \emptyset; Q := \emptyset$ 
3  $C_{trg} := C_{pr}$ 
  -----
4 while ( $true$ )  $\{$ 
5   if ( $Q = \emptyset$ )  $\{$ 
6      $(v, b) := Assgn(F_1, F_2, X)$ 
7      $UpdQueue(Q \parallel v, b)$ 
8      $(ans, C', S') := BCP(\eta \parallel \xi)$ 
9     if ( $ans = NoBcktr$ )
10      continue
  -----
11  $(S, C) := Lrn(ans, \phi, C', S')$ 
12  $Store(F_1, F_2, Ds \parallel S, C)$ 
13 if ( $C_{trg} = C_{pr}$ )  $\{$ 
14    $RegBcktr(\vec{a}, T \parallel S, C)$ 
15   if ( $Cond(S) = \emptyset$ ) return
16    $UpdQueue(Q \parallel \vec{a}, S, C)$ 
17   continue
  -----
18*  $SpecBcktr(\vec{a}, T \parallel S, C)$ 
19* if ( $\neg TrgDone(T)$ )  $\{$ 
20*    $UpdQueue(Q \parallel \vec{a}, S, C)$ 
21*   continue
22*  $(C_{trg}, S) := NewTrg(\phi \parallel)$ 
23* if ( $C_{trg} \neq C_{pr}$ ) continue
24* if ( $Cond(S) = \emptyset$ ) return
25*  $UpdQueue(Q \parallel \vec{a}, S)$ 

```

Fig. 2. The $PrvRed$ procedure

generating a conflict clause C or a new D-sequent S for C_{trg} (line 11). Then $PrvRed$ stores S in Ds (if it is worth reusing) or adds C to $F_1 \wedge F_2$. If a clause of F_1 is used in generation of C , the latter is added to F_1 . Otherwise, C is added to F_2 . If the current target is C_{pr} , one uses “regular” backtracking (lines 13-14, see Subsection XI-E). If the conditional of S is empty, $PrvRed$ terminates (line 15). Otherwise, an assignment derived from S or C is added to \vec{a} (line 16). This derivation is possible because after backtracking, the generated conflict clause C (or the D-sequent S) becomes *unit*. If the assignment above is derived from S , $PrvRed$ keeps S until the decision level of this assignment is eliminated (even if S is not stored in Ds). The third part (lines 18-25) is described in Section XII.

⁷The reason for making decision assignments on variables of Y before those of X is as follows. The final goal of PQE is to derive clauses $C(Y)$ making F_1 redundant in $\exists X[F_1 \wedge F_2]$. Giving preference to variables of Y simplifies generation of such clauses. If a conflict occurs at a decision level started by an assignment to variable $v \in Y$, one can easily derive a conflict clause depending only on variables of Y .

The operation of $PrvRed$ in the while loop can be partitioned into three parts identified by dotted lines. The first part (lines 5-10) starts with checking if the assignment queue Q is empty. If so, a new assignment $v = b$ is picked (line 6) where $v \in (X \cup Y)$ and $b \in \{0, 1\}$ and added to Q . $PrvRed$ first assigns⁷ the variables of Y . So $v \in X$, only if all variables of Y are assigned. Then $PrvRed$ calls the BCP procedure. If BCP identifies a backtracking condition, $PrvRed$ goes to the second part. (This means that C_{trg} is proved redundant in the subspace \vec{a} . In particular, a backtracking condition is met if BCP falsifies a clause C' or activates a D-sequent S' learned earlier.) Otherwise, $PrvRed$ begins a new iteration.

$PrvRed$ starts the second part (lines 11-17) with generating a conflict clause C or a new D-sequent S for C_{trg} (line 11). Then $PrvRed$ stores S in Ds (if it is worth reusing) or adds C to $F_1 \wedge F_2$. If a clause of F_1 is used in generation of C , the latter is added to F_1 . Otherwise, C is added to F_2 . If the current target is C_{pr} , one uses “regular” backtracking (lines 13-14, see Subsection XI-E). If the conditional of S is empty, $PrvRed$ terminates (line 15). Otherwise, an assignment derived from S or C is added to \vec{a} (line 16). This derivation is possible because after backtracking, the generated conflict clause C (or the D-sequent S) becomes *unit*. If the assignment above is derived from S , $PrvRed$ keeps S until the decision level of this assignment is eliminated (even if S is not stored in Ds). The third part (lines 18-25) is described in Section XII.

C. BCP

The main loop of *BCP* consists of the three parts shown in Fig. 3 by dotted lines. (Parameters η and ξ are defined in Fig. 2.) In the first part (lines 2-9), *BCP* extracts an assignment $w = b$ from the assignment queue Q (line 2). It can be a decision assignment or one derived from a clause C or D-sequent S . Then, *BCP* updates the current assignment \vec{a} (line 9). Lines 3-8 are explained in Subsection XII-B.

```

BCP( $\eta \parallel \xi$ ) {
1 while ( $Q \neq \emptyset$ ) {
2   ( $w, b, C, S$ ) := Pop( $Q$ )
3*  if ( $C = C_{trg}$ ) {
4*    $C' := BCP^*(\eta \parallel \xi, w, b)$ 
5*    $C_{trg} := NewTrg(T)$ 
6*   if ( $C' \neq nil$ )
7*     return( $FlsCls, C', nil$ )
8*   break; }
9   UpdAssgn( $\vec{a} \parallel w, b, C, S$ )
-----
10  if ( $Satisf(C_{trg}, w, b)$ )
11    return( $SatTrg, nil, nil$ )
12   $C' := ChkCls(Q \parallel F_1, F_2, w, b)$ 
13  if ( $C' \neq nil$ )
14    return( $FlsCls, C', nil$ )
15   $S' := ChkDsq(Q \parallel Ds, w, b)$ 
16  if ( $S' \neq nil$ )
17    return( $ActDseq, nil, S'$ )
-----
18  if ( $Blocked(C_{trg}, \vec{a}, F_1, F_2)$ )
19    return( $BlkTrg, nil, nil$ )
20  return( $NoBcktr, nil, nil$ )

```

Fig. 3. The *BCP* procedure

added to Q (see Subsection VIII-A). If an active D-sequent S' is found, *BCP* terminates (lines 16-17).

Finally, *BCP* checks if C_{trg} is blocked (lines 18-19). If not, *BCP* reports that no backtracking condition is met (line 20).

D. D-sequent generation

When *BCP* reports a backtracking condition, the *Lrn* procedure (line 11 of Fig 2) generates a conflict clause C or a D-sequent S . *Lrn* generates a conflict clause when *BCP* returns a falsified clause C' and every implied assignment used by *Lrn* to construct C is derived from a clause [33]. Adding C to $F_1 \wedge F_2$ makes the current target clause C_{trg} redundant in subspace \vec{a} . Otherwise⁸, *Lrn* generates a D-sequent S for C_{trg} . The D-sequent S is built similarly to a conflict clause C . First, *Lrn* forms an initial D-sequent S equal to $(\vec{q}, H) \rightarrow C_{trg}$ (unless an existing D-sequent is activated by *BCP*). The conditional \vec{q} and structure constraint H of S depend on the backtracking condition returned by *BCP*. If \vec{q} contains assignments *derived* at the current decision level, *Lrn* tries to get rid of them as it is done by a SAT-solver generating a conflict clause. Only instead of resolution, *Lrn* uses the join operation. Let $w = b$ be the assignment of \vec{q} derived at the current decision level where $b \in \{0, 1\}$. If it is

⁸There is one case where *Lrn* generates a D-sequent *and* a clause (see Appendix III-E).

derived from a D-sequent S' equal to $(\vec{q}', H') \rightarrow C_{trg}$, *Lrn* joins S and S' at w to produce a new D-sequent S . If $w = b$ is derived from a clause B , *Lrn* joins S with the atomic D-sequent S' of the second kind stating the redundancy of C_{trg} when B is falsified. S' is equal to $(\vec{q}', H') \rightarrow C_{trg}$ where \vec{q}' is the shortest assignment falsifying B and $H' = \{B\}$. *Lrn* keeps joining D-sequents until it builds a D-sequent S whose conditional does not contain assignments derived at the current decision level (but may contain the *decision* assignment of this level). Appendix III gives examples of D-sequents built by *Lrn*.

E. Regular backtracking

If C_{trg} is the primary target C_{pr} , *PrvRed* calls the backtracking procedure *RegBcktr* (line 14 of Fig. 2). If *Lrn* returns a conflict clause C , *RegBcktr* backtracks to the smallest decision level where C is still unit. So an assignment can be derived from C . (This is how a SAT-solver with conflict clause learning backtracks [33].) Similarly, if *Lrn* returns a D-sequent S , *RegBcktr* backtracks to the smallest decision level where S is still unit. So an assignment can be derived from S .

XII. USING SECONDARY-TARGET CLAUSES

The objective of *PrvRed* (see Fig. 2) is to prove the primary target clause C_{pr} redundant. To achieve this goal, *PrvRed* may need to prove redundancy of so-called secondary target clauses. In this section, we describe how this is done.

A. The reason for using secondary targets

Let $\exists X[F_1(X, Y) \wedge F_2(X, Y)]$ be an \exists CNF formula. Assume that *PrvRed* tries to prove redundancy of the clause $C_{pr} \in F_1$ where $C_{pr} = y_1 \vee x_2$. Suppose that \vec{a} is the current assignment to $X \cup Y$ and y_1 is assigned 0 in \vec{a} whereas x_2 is not assigned yet. Since C_{pr} is falsified in subspace $\vec{a} \cup \{x_2 = 0\}$, the assignment $x_2 = 1$ is derived by *BCP*. However, the goal of *PrvRed* is to prove C_{pr} *redundant* rather than satisfy $F_1 \wedge F_2$. The fact that C_{pr} is falsified in a subspace says nothing about its redundancy in this subspace.

To address the problem above, *PrvRed* explores the subspace $\vec{a} \cup \{x_2 = 1\}$ to prove redundancy of the clauses of $F_1 \wedge F_2$ *resolvable* with C_{pr} on x_2 . These clauses are called *secondary targets*. Proving their redundancy results in proving redundancy of C_{pr} . If $F_1 \wedge F_2$ is *unsatisfiable* in the subspace $\vec{a} \cup \{x_2 = 1\}$, *PrvRed* generates a conflict clause that does not depend on x_2 . Adding it to $F_1 \wedge F_2$ makes C_{pr} redundant in the subspace \vec{a} and an atomic D-sequent of the second kind is built. If $F_1 \wedge F_2$ is *satisfiable* in the subspace $\vec{a} \cup \{x_2 = 1\}$, *PrvRed* simply proves redundancy of the secondary targets. Then C_{pr} is *blocked* at variable x_2 and an atomic D-sequent of the third kind is generated stating the redundancy of C_{pr} in the subspace \vec{a} .

The same strategy is used for *every current* target clause C_{trg} (secondary or primary). Whenever C_{trg} becomes unit, *PrvRed* generates new secondary targets to be proved redundant. These are the clauses of $F_1 \wedge F_2$ *resolvable* with C_{trg} on the variable that is currently unassigned in C_{trg} .

B. Generation of secondary targets

To keep track of secondary targets *PrvRed* maintains a stack T of target levels. (Appendix IV gives an example of how T is updated.) The bottom level of T consists of the primary target clause C_{pr} . All other levels are meant for secondary targets. Every such a level is specified by a pair (C, w) where C is either C_{pr} or a secondary target clause and $w \in X$ is a variable of C . They are called the **key clause** and the **key variable** of this level. The secondary targets specified by this level are the clauses of $F_1 \wedge F_2$ resolvable with C on w . The top level of T specifies the current target clause C_{trg} . Namely, C_{trg} is resolvable with the key clause C on the key variable w of the top level of T . Once C_{trg} is proved redundant, another clause resolvable with C on w and not proved redundant yet is chosen as the new target.

BCP picks assignment $w = b$ derived from C_{trg} only if Q does not contain any other assignments (line 2 of Fig. 3). Lines (4-8) show what happens next. First, the *BCP** procedure is called to make the assignment $w = b$. *BCP** is similar to *BCP* of a SAT-solver: it derives assignments only from clauses and returns a falsified clause C' if a conflict occurs. The only difference is that every time *BCP** finds a unit clause C , a new target level of T is generated. (The reason is that every unit clause produced by *BCP** is resolvable either with C_{trg} or with some secondary target generated by *BCP**.) Let w be the unassigned variable of C . Then this level is specified by the pair (C, w) . It consists of the clauses of $F_1 \wedge F_2$ resolvable with C on w . On completing *BCP**, a new C_{trg} is chosen among the clauses of the top level of T (line 5). If a conflict occurred during *BCP**, the *BCP* procedure terminates (lines 6-7). Otherwise, the main loop of *BCP* terminates (line 8).

C. Special backtracking

PrvRed uses a special backtracking procedure called *SpecBcktr* (line 18 of Fig. 2) if the current target is not the primary target C_{pr} . (An example of special backtracking is given in Appendix V.) If *Lrn* returns a conflict clause C (line 11), *SpecBcktr* backtracks in the same manner as *RegBcktr* (line 14). Namely, it jumps to the smallest decision level where C is still unit. The difference is that *SpecBcktr* also eliminates the target levels of T that are jumped over. Namely, if the key variable w of a target level is unassigned by *SpecBcktr*, this level is eliminated. (Adding C makes all X -clauses of $F_1 \wedge F_2$ redundant in the current subspace. So proving redundancy of secondary targets with variable w is not needed anymore.)

Suppose *Lrn* returns a D-sequent S . Since S states redundancy of C_{trg} , the scope of *SpecBcktr* is limited to the variables assigned after the key variable w of the top target level of T (to which C_{trg} belongs). If some variables assigned after w remain assigned on completing *SpecBcktr*, proving C_{trg} redundant is not over yet. In this case, *PrvRed* adds the assignment derived from S to the queue Q and starts the next iteration of the while loop (lines 19-21 of Fig. 2). Otherwise, C_{trg} is proved redundant up to the point of origin and *PrvRed* calls *NewTrg* to look for a new target (line 22). Namely, *NewTrg* looks for a clause resolvable with the key

clause C on the key variable w of the top level of T that is not proved redundant yet.

If *NewTrg* fails to find a target in the top level of T , C is blocked at w . Then *NewTrg* generates a D-sequent for C and deletes the top level of T . This entails returning the clauses of this level proved redundant back in $F_1 \wedge F_2$ and unassigning w . Then *NewTrg* looks for a target in the new top level of T and so on. If *NewTrg* finds C_{trg} that is not the primary target C_{pr} , *PrvRed* starts a new iteration of the while loop (line 23). Otherwise, *NewTrg* sets C_{trg} to C_{pr} . It also returns a D-sequent S for C_{pr} since C_{pr} is blocked. If the conditional of S is empty, C_{pr} is redundant unconditionally and *PrvRed* terminates (line 24). Otherwise, the assignment derived from S is added to Q and a new iteration begins (line 25).

XIII. EXPERIMENTAL RESULTS

In this section, we evaluate an implementation of *DS-PQE*⁺. (Appendix VIII provides more experimental data). Our preliminary experiments showed that structure constraints can grow very large, which makes storing D-sequents expensive. In Appendix VII, we discuss various methods of dealing with this problem. In our experiments, we used the following idea. One can reduce the size of structure constraints by storing/reusing only D-sequents for the target clauses of k bottom levels of the stack T . In particular, one can safely reuse the D-sequents for the primary target clause ($k = 0$) without computing structure constraints at all (see Appendix VII).

For the evaluation of *DS-PQE*⁺, we use Circuit-SAT, the first problem listed in Section III. We consider this problem in the form repeatedly solved in IC3 [7]: given a state \bar{z} , find the states from which \bar{z} can be reached in one transition. In [37], it was suggested to look for the largest subset of these states forming a *cube*. In this section, we use a variation of this problem for evaluation of *DS-PQE*⁺. We demonstrate that *DS-PQE*⁺ dramatically outperforms *DS-PQE* of [25].

We also compare *DS-PQE*⁺ with two SAT-based methods. On examples with deterministic Transition Relations (TRs), both methods are faster than *DS-PQE*⁺. However, method 1 shows poorer results (in terms of cube size). Method 2 is comparable with *DS-PQE*⁺ in terms of cube size but is, in general, inapplicable to *non-deterministic* TRs. (A deterministic TR is specified by a deterministic circuit N i.e. an input to N produces only one output. A deterministic TR can become non-deterministic e.g. after pre-processing [15] performed to speed up the SAT-checks of IC3). Importantly, no optimization techniques are used in our implementation of *DS-PQE*⁺ yet. So its performance can be dramatically improved.

Let $N(X, Y, Z)$ be a combinational circuit where X, Y and Z are sets of input, internal and output variables respectively. Let \bar{z} be a full assignment to Z . The problem we consider is to find the input assignments for which N evaluates to \bar{z} . Let $C_{\bar{z}}$ be the longest clause falsified by \bar{z} . Let $F(X, Y, Z)$ be a CNF formula specifying N . Let W denote $Y \cup Z$. As we mentioned in Subsection III-A, the problem above reduces to finding $G(X)$ such that $G \wedge \exists W[F] \equiv \exists W[C_{\bar{z}} \wedge F]$ i.e. to PQE. We assume here that N produces at least one output

for every input. So, $\exists W[F] \equiv 1$ and $G \equiv \exists W[C_{\vec{z}} \wedge F]$. If $C \in G$, then \overline{C} specifies a cube of input assignments for which N evaluates to \vec{z} . So, a shorter clause C specifies a larger set of input assignments producing output \vec{z} .

TABLE I
TAKING $C_{\vec{z}}$ OUT OF THE SCOPE OF QUANTIFIERS IN $\exists W[C_{\vec{z}} \wedge F]$. THE TIME LIMIT IS 100 SECONDS.

name	#inps	$DS-PQE$		$DS-PQE^+$ no learning		$DS-PQE^+$ limited learning	
		#dseqs $\times 10^3$	time (s.)	#dseqs $\times 10^3$	time (s.)	#dseqs $\times 10^3$	time (s.)
pdtvisheap00	37	231	3.4	0.9	0.03	0.1	0.02
texasifetch1p1	87	>416	*	6.8	0.4	0.7	0.4
pdtpmrsretherrf	93	>10,128	*	17	1	0.6	0.05
pdtvisblackjack2	109	>6,857	*	690	61	226	18
pdtvisvsar04	147	2,507	68	6.5	0.9	0.4	0.06
texaspimainp01	253	>35.6	*	71	30	8.7	3.4
eijkbs3330	286	>17	*	2.6	0.7	0.2	0.2
nusmvtcasp2	325	>3,301	*	22	1.4	3.4	0.2
pdtvisfeistel	429	>2,305	*	>44	*	7.3	13
pdtpmvsal6a	453	>1,742	*	171	87	1.1	1

First, we compared the performance of $DS-PQE$ [25], $DS-PQE^+$ with no learning and $DS-PQE^+$ with limited learning (only for primary targets). We used the transition relation of a HWMCC-10 benchmark as circuit N . For the sake of simplicity, we ignored the difference between latched and combinational input variables of N . (In the context of model checking, the literals of combinational variables are supposed to be *dropped* from $C \in G$ to make \overline{C} a cube of *states*.) In Table I, we give a sample of the set of benchmarks we tried that shows the general trend. The first column gives the name of a benchmark. The second column shows the number of input variables of N . The remaining columns give the number of generated D-sequents (in thousands) and the run time for each PQE procedure. Table I shows that $DS-PQE^+$ without learning outperforms $DS-PQE$ due to generating fewer D-sequents. For the same reason, $DS-PQE^+$ with learning outperforms $DS-PQE^+$ without learning.

TABLE II
COMPARISON WITH SAT-BASED METHODS (DETERMINISTIC CIRCUITS)

name	#inps	SAT method 1		SAT method 2		$DS-PQE^+$ limited learning	
		#len- gth	time (s.)	#len- gth	time (s.)	#len- gth	time (s.)
visemodel	26	22	0.1	6	0	6	0.01
bobcohdoptdcd4	62	53	0.1	46	0.1	42	0.1
eijkbs3330	286	155	0.2	43	0.1	43	0.2
pdtvisfeistel	429	363	1.3	1	0.3	1	4.8
139464p0	1,002	992	3	625	3.3	572	23

The formula $G(X)$ above can also be found by SAT. We tried two SAT-based methods using Minisat [16] as a SAT-solver. Method 1 (inspired by [34]) is essentially a *universal* QE algorithm whereas method 2 is applicable only for formulas derived from *deterministic* circuits. Method 1 looks for an assignment $(\vec{x}, \vec{y}, \vec{z})$ satisfying $G \wedge F \wedge U_{\vec{z}}$. (Originally, $G = \emptyset$.) Here $U_{\vec{z}}$ is the set of unit clauses specifying \vec{z} . Then it builds the smallest assignment $(\vec{x}', \vec{y}, \vec{z})$ where $\vec{x}' \subseteq \vec{x}$ that still satisfies $G \wedge F \wedge U_{\vec{z}}$. Finally, the longest clause $C_{\vec{x}'}$ falsified by \vec{x}' is added to G and a new satisfying assignment

is generated. Method 1 terminates when $G \wedge F \wedge U_{\vec{z}}$ is unsatisfiable. Method 2 follows the idea employed in advanced implementations of IC3/PDR [37], [11]. Namely, it uses the satisfying assignment $(\vec{x}, \vec{y}, \vec{z})$ above to build formula R equal to $U_{\vec{x}} \wedge F \wedge G \wedge C_{\vec{z}}$. Here $U_{\vec{x}}$ is the set of unit clauses specifying \vec{x} . If circuit N is *deterministic*, R is unsatisfiable. Then one extracts the subset of $U_{\vec{x}}$ used in the proof of unsatisfiability of R . The clause made up of the negated literals of this subset is added to G . Then a new satisfying assignment is generated (if any).

TABLE III
NON-DETERMINISTIC CIRCUITS

name	SAT method 1		$DS-PQE^+$ limited learning	
	#len- gth	time (s.)	#len- gth	time (s.)
visemodel	14	0.04	6	0.01
bob.ptdcd4	47	0.1	42	0.1
eijkbs3330	75	0.2	43	0.2
pdt.sfeistel	140	1.1	1	13
139464p0	748	3.4	577	32

Table II compares the SAT-based methods above with $DS-PQE^+$ (with limited learning) on 5 formulas showing the general trend. All three methods were run until 1,000 clauses of G were generated or the problem was finished.

We computed the length of the shortest clause generated by each method and its run time. Table II shows that the SAT-based methods are faster than $DS-PQE^+$ whereas the best clause generated by method 2 and $DS-PQE^+$ is shorter than that of method 1. So method 2 is the winner.

In Table III we repeat the same experiment for *non-deterministic* versions of circuits from Table II. To make the original circuit N non-deterministic, we just dropped a fraction of clauses in the formula F representing N . A non-deterministic circuit N may produce different outputs for the same input. In this case, the formula R above can be satisfiable, which renders method 2 *inapplicable*. Table III shows that method 1 is still faster than $DS-PQE^+$ but generates much longer clauses.

XIV. SOME BACKGROUND

In this section, we give some background on learning in branching algorithms⁹ used in verification. For such algorithms, it is important to share information obtained in different subspaces. An important example of such sharing is the identification of isomorphic subgraphs when constructing a BDD [9]. Another example is SAT-solving with conflict driven learning [33], [36], [16]. The difference between learning in BDDs/SAT-solvers and D-sequents is that the former is semantic¹⁰ whereas the latter is structural.

The appeal of finding structural properties is that they are *formula-specific*. So using such properties can give a dramatic performance improvement. An obvious example of a structural property is symmetry. In [12], [2], [14], the permutational symmetry of a CNF formula F is exploited via adding

⁹Information about algorithms performing *complete* QE for propositional logic can be found in [9], [10] (BDD based) and [34], [29], [17], [28], [8], [30], [6], [5] (SAT-based).

¹⁰A BDD of a formula is just a compact representation of its truth table. A conflict clause C is implied by the formula F from which C is derived and implication is a semantic property of F .

“structural implications”. By a structural implication of F , we mean a clause C that, in general, is not implied by F but preserves the *equisatisfiability* of $F \wedge C$ to F . For instance, to keep only one satisfying assignment (if any) out of a set of *symmetric* ones, symmetry-breaking clauses are added to F .

ATPG is another area where formula structure is exploited. In ATPG methods [1], one reasons about a circuit in terms of *signal propagation*. In the classic paper [32], signal propagation is simulated in a CNF formula F generated for identifying a circuit fault. Formula F specifies the functionality of correct and faulty circuits. Additional variables and clauses are added to F to facilitate signal reasoning. These extra clauses, like in formulas with symmetries, are “structural implications”.

The difference between D-sequents and traditional methods of exploiting formula structure is twofold. First, redundancy is a *very general* structural property. For that reason, the machinery of D-sequents can be applied to *any CNF formula* (e.g. a random CNF formula). Second, a traditional way to take into account structure is to add some kind of structural implications and then run a verification engine performing *semantic* derivations (e.g. a SAT-solver). The machinery of D-sequents is different in that it performs structural derivations (namely, proving redundancy of clauses with quantified variables) *all the way* until some semantic fact is established e.g. $F_1^* \wedge \exists X[F_2] \equiv \exists X[F_1 \wedge F_2]$.

Removal of redundant clauses is used in preprocessing procedures of QBF-algorithms and SAT-solvers [15], [4]. Redundant clauses are also identified in the inner loop of SAT-solving (inprocessing) [27]. These procedures identify *unconditional* clause redundancies by recognizing some situations where such redundancies can be easily proved.

XV. CONCLUSIONS

We consider Partial Quantifier Elimination (PQE) on propositional CNF formulas with existential quantifiers. In PQE, only a (small) subformula is taken out of the scope of quantifiers. The appeal of PQE is that in many verification problems one can use PQE instead of *complete* QE and the former can be dramatically more efficient. Earlier, we developed a PQE algorithm based on the notion of clause redundancy. Since redundancy is a structural property, reusing learned information is not trivial. In this paper, we provide some theory addressing this problem. Besides, we introduce a new PQE algorithm that performs single-event backtracking. This algorithm bears some similarity to a SAT-solver and facilitates reusing learned information. We show experimentally that the new PQE algorithm is dramatically faster than its predecessor. We believe that reusing learned information is an important step in making PQE practical.

XVI. DIRECTIONS FOR FUTURE RESEARCH

In our future research we are planning to focus on the following two directions. First, although $DS-PQE^+$ shows an obvious improvement over $DS-PQE$, it is too complex. So, we will try to find a simpler version of $DS-PQE^+$ that preserves its good performance. Second, we want to relax the

decision making constraint (quantified variables are assigned before unquantified). As we know from the practice of SAT-solving, a poor choice of branching variables may lead to a significant performance degradation.

REFERENCES

- [1] M. Abramovici, M. Breuer, and A. Friedman. *Digital Systems Testing and Testable Design*. John Wiley & Sons, 1994.
- [2] F. Aloul, K. Sakallah, and I. Markov. Efficient symmetry breaking for boolean satisfiability. *IEEE Transactions on Computers*, 55(5):549–558, May 2006.
- [3] A. Biere, A. Cimatti, E. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using sat procedures instead of bdds. In *DAC*, pages 317–320, 1999.
- [4] A. Biere, F. Lonsing, and M. Seidl. Blocked clause elimination for qbf. *CADE-11*, pages 101–115, 2011.
- [5] N. Björner and M. Janota. Playing with quantified satisfaction. In *LPAR*, 2015.
- [6] N. Björner, M. Janota, and W. Klieber. On conflicts and strategies in qbf. In *LPAR*, 2015.
- [7] A. R. Bradley. Sat-based model checking without unrolling. In *VMCAI*, pages 70–87, 2011.
- [8] J. Brauer, A. King, and J. Kriener. Existential quantification as incremental sat. *CAV-11*, pages 191–207, 2011.
- [9] R. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.
- [10] P. Chauhan, E. Clarke, S. Jha, J.H. Kukula, H. Veith, and D. Wang. Using combinatorial optimization methods for quantification scheduling. *CHARME-01*, pages 293–309, 2001.
- [11] H. Chockler, A. Ivrii, A. Matsliah, S. Moran, and Z. Nevo. Incremental formal verification of hardware. In *FMCAD-11*, pages 135–143, 2011.
- [12] J. Crawford, M. Ginsberg, E. Luks, and A. Roy. Symmetry-breaking predicates for search problems. In *Int. Conf. on Principles of Knowledge Representation and Reasoning*, pages 148–159, 1996.
- [13] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5(7):394–397, July 1962.
- [14] J. Devriendt, B. Bogaerts, M. Bruynooghe, and M. Denecker. Improved static symmetry breaking for sat. In *SAT*, 2016.
- [15] N. Eén and A. Biere. Effective preprocessing in sat through variable and clause elimination. In *SAT*, pages 61–75, 2005.
- [16] N. Eén and N. Sörensson. An extensible sat-solver. In *SAT*, pages 502–518, Santa Margherita Ligure, Italy, 2003.
- [17] M. Ganai, A. Gupta, and P. Ashar. Efficient sat-based unbounded symbolic model checking using circuit cofactoring. *ICCAD-04*, pages 510–517, 2004.
- [18] E. Goldberg. Equivalence checking by logic relaxation. Technical Report arXiv:1511.01368 [cs.LO], 2015.
- [19] E. Goldberg. Equivalence checking by logic relaxation. In *FMCAD-16*, pages 49–56, 2016.
- [20] E. Goldberg. Property checking without invariant generation. Technical Report arXiv:1602.05829 [cs.LO], 2016.
- [21] E. Goldberg. Quantifier elimination with structural learning. Technical Report arXiv: 1810.00160 [cs.LO], 2018.
- [22] E. Goldberg and P. Manolios. Quantifier elimination by dependency sequents. In *FMCAD-12*, pages 34–44, 2012.
- [23] E. Goldberg and P. Manolios. Removal of quantifiers by elimination of boundary points. Technical Report arXiv:1204.1746 [cs.LO], 2012.
- [24] E. Goldberg and P. Manolios. Quantifier elimination via clause redundancy. In *FMCAD-13*, pages 85–92, 2013.
- [25] E. Goldberg and P. Manolios. Partial quantifier elimination. In *Proc. of HVC-14*, pages 148–164. Springer-Verlag, 2014.
- [26] E. Goldberg and P. Manolios. Software for quantifier elimination in propositional logic. In *ICMS-2014, Seoul, South Korea, August 5-9*, pages 291–294, 2014.
- [27] M. Järvisalo, M. Heule, and A. Biere. Inprocessing rules. *IJCAR-12*, pages 355–370, 2012.
- [28] J. Jiang. Quantifier elimination via functional composition. In *Proceedings of the 21st International Conference on Computer Aided Verification*, CAV-09, pages 383–397, 2009.
- [29] H. Jin and F. Somenzi. Prime clauses for fast enumeration of satisfying assignments to boolean circuits. *DAC-05*, pages 750–753, 2005.

- [30] W. Klieber, M. Janota, J. Marques-Silva, and E. Clarke. Solving qbf with free variables. In *CP*, pages 415–431, 2013.
- [31] O. Kullmann. New methods for 3-sat decision and worst-case analysis. *Theor. Comput. Sci.*, 223(1-2):1–72, 1999.
- [32] T. Larrabee. Test pattern generation using boolean satisfiability. *IEEE Trans. on Comp.-Aided Design of Integr. Circuits and Systems*, 11(1):4–15, Jan 1992.
- [33] J. Marques-Silva and K. Sakallah. Grasp – a new search algorithm for satisfiability. In *ICCAD-96*, pages 220–227, 1996.
- [34] K. McMillan. Applying sat methods in unbounded symbolic model checking. In *Proc. of CAV-02*, pages 250–264. Springer-Verlag, 2002.
- [35] K. McMillan. Interpolation and sat-based model checking. In *CAV-03*, pages 1–13. Springer, 2003.
- [36] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: engineering an efficient sat solver. In *DAC-01*, pages 530–535, New York, NY, USA, 2001.
- [37] E. Niklas, A. Mishchenko, and R. Brayton. Efficient implementation of property directed reachability. In *FMCAD-11*, pages 125–134, 2011.

APPENDIX I

REDUNDANCY OF A CLAUSE IN A SUBSPACE

In this appendix, we discuss the following problem. Let $\exists X[F]$ be an \exists CNF formula. Let C be an X -clause of F redundant in $\exists X[F]$ in subspace \vec{q} . Let \vec{r} be an assignment to $\text{Vars}(F)$ where $\vec{q} \subset \vec{r}$. Intuitively, the clause C should remain redundant in the *smaller* subspace specified by \vec{r} . However, this is not the case. $\exists X[F_{\vec{q}}] \equiv \exists X[F_{\vec{q}} \setminus \{C_{\vec{q}}\}]$ does not imply $\exists X[F_{\vec{r}}] \equiv \exists X[F_{\vec{r}} \setminus \{C_{\vec{r}}\}]$ (see the example below).

Example 6: Let $G(X)$ be a satisfiable formula. Then every $G^* \subseteq G$ is redundant in $\exists X[G]$. Assume that G is unsatisfiable in subspace \vec{r} . Then there is $G^* \subseteq G$ that is *not* redundant in $\exists X[G]$ in subspace \vec{r} . So G^* is redundant in subspace $\vec{q} = \emptyset$ and is not redundant in subspace \vec{r} where $\vec{q} \subset \vec{r}$. An obvious problem here is that the formula $G_{\vec{r}}$ does not preserve the structure of G (in terms of redundancy of clauses).

The problem above can be easily addressed by using a more sophisticated notion of redundancy called *virtual redundancy* [21]. (The latter is different from redundancy specified by Definition 9.) However, this would require adding more definitions and propositions. So, for the sake of simplicity, in this paper, we assume that if C is redundant in $\exists X[F]$ in a subspace \vec{q} (where redundancy is specified by Definition 9), it is also redundant in subspace \vec{r} if $\vec{q} \subset \vec{r}$ holds.

APPENDIX II

GENERATION OF NEW D-SEQUENTS BY SUBSTITUTION

In Section X, we recalled two methods of producing new D-sequents. In this appendix, we describe one more procedure for generating a new D-sequent. This procedure is to reshape the structure constraint H of a D-sequent by substituting a clause $C \in H$ with the structure constraint of another D-sequent stating redundancy of C .

Proposition 6: Let $\exists X[F]$ be an \exists CNF formula. Let $(\vec{q}', H') \rightarrow C'$ and $(\vec{q}'', H'') \rightarrow C''$ be consistent D-sequents (see Definition 13). Let C'' be in H' . Then the D-sequent $(\vec{q}, H) \rightarrow C'$ holds where $\vec{q} = \vec{q}' \cup \vec{q}''$, $H = (H' \setminus \{C''\}) \cup H''$.

The proposition below is an implication of Proposition 6 to be used in Appendix VII. Since this proposition is not mentioned in [21], we prove it here.

Proposition 7: Let $\exists X[F]$ be an \exists CNF formula. Let S be a D-sequent equal to $(\vec{q}, H) \rightarrow C$. Let \vec{r} be an assignment satisfying at least one clause of H where $\vec{q} \subset \vec{r}$ holds. Then the D-sequent S' equal to $(\vec{r}, H') \rightarrow C$ holds where H' is obtained from H by removing the clauses satisfied by \vec{r} .

Proof: Let C_1, \dots, C_k be the clauses of H satisfied by \vec{r} . For each clause C_i , $1 \leq i \leq k$, one can build an atomic D-sequent S_i of the first kind equal to $(\vec{q}_i, H_i) \rightarrow C_i$. Here $H_i = \emptyset$ and $\vec{q}_i = (v_i = b_i)$ is an assignment satisfying C_i where $\vec{q}_i \subseteq \vec{r}$. It is not hard to show that S, S_1 are consistent (see Definition 13). By applying Proposition 6, one obtains a new D-sequent S' equal to $(\vec{q}', H') \rightarrow C$ where $\vec{q}' = \vec{q} \cup \vec{q}_1$ and $H' = H \setminus \{C_1\}$. Since the new D-sequent S' is consistent with S_2 , one can apply Proposition 6 again. Going on in such a manner, one obtains the D-sequent S' equal to $(\vec{r}', H') \rightarrow C$ where $\vec{r}' = \vec{q} \cup \vec{q}_1 \dots \cup \vec{q}_k$ and $H' = H \setminus \{C_1, \dots, C_k\}$. Since $\vec{r}' \subseteq \vec{r}$, the D-sequent $(\vec{r}, H') \rightarrow C$ holds as well (see Appendix I) ■

APPENDIX III

EXAMPLES OF D-SEQUENT GENERATION BY *Lrn*

In this appendix, we give examples of how *Lrn* builds D-sequents. (The only exception is Appendix III-A where we describe how a conflict clause generated.) We continue using the notation of Section XI. In particular, we assume that *DS-PQE+* is applied to take F_1 out of the scope of quantifiers in $\exists X[F_1(X, Y) \wedge F_2(X, Y)]$.

A. Generation of a conflict clause

Suppose that a clause $C_{f_{ls}}$ of $F_1 \wedge F_2$ is falsified in *B_{CP}* (see Fig. 3). Let \vec{a} be the current assignment at the point of *B_{CP}* where $C_{f_{ls}}$ gets falsified. (So \vec{a} falsifies $C_{f_{ls}}$.) Then *Lrn* generates a *conflict clause* i.e. a clause falsified by \vec{a} in which there is only one literal falsified at the conflict level. This clause is built as follows [33]. First, *Lrn* picks the last literal *Lit* of $C_{f_{ls}}$ falsified by \vec{a} . *Lrn* takes the clause C from which the value falsifying *Lit* was derived and resolves it with $C_{f_{ls}}$ producing a new clause $C_{f_{ls}}$ falsified by \vec{a} . Then *Lrn* again picks the last literal *Lit* of $C_{f_{ls}}$ falsified by \vec{a} . This goes on until only one literal of $C_{f_{ls}}$ is falsified by an assignment made at the conflict level and this is the *decision* assignment of this level. At this point, $C_{f_{ls}}$ is a conflict clause that is added to $F_1 \wedge F_2$.

B. D-sequent generation when current target is satisfied

Suppose that $Y = \{y\}$ and $F_1 \wedge F_2$ contains (among others) the clauses $C_1 = y \vee x_1$, $C_2 = x_1 \vee x_2$. Suppose C_2 is the current target clause and *PrvRed* makes the decision assignment $y = 0$. *B_{CP}* finds out that C_1 is unit and derives the assignment $x_1 = 1$ satisfying C_2 . So *B_{CP}* terminates reporting the backtracking condition *SatTrg* (line 11 of Fig. 3). At this point the current assignment \vec{a} is equal to $(y = 0, x_1 = 1)$. Then *Lrn* builds a D-sequent S as follows. It starts with the atomic D-sequent S of the first kind equal to $(\vec{q}, H) \rightarrow C_2$ where $\vec{q} = (x_1 = 1)$ and $H = \emptyset$. The D-sequent S states that C_2 is satisfied by \vec{q} and hence redundant in the subspace

\vec{q} (and so in the subspace \vec{a}). The conditional \vec{q} contains the assignment ($x_1 = 1$) derived from clause C_1 . *Lrn* gets rid of this assignment as described in Subsection XI-D. First, it forms the D-sequent S' equal to $(\vec{q}', H') \rightarrow C_2$ where $\vec{q}' = (y = 0, x_1 = 0)$ and $H' = \{C_1\}$. This is an atomic D-sequent of the second kind stating the redundancy of C_2 in the subspace where C_1 is falsified. Then *Lrn* joins S and S' at variable x_1 to obtain a new D-sequent S equal to $(\vec{q}, H) \rightarrow C_2$ where $\vec{q} = (y = 0)$ and $H = \{C_1\}$. The conditional \vec{q} of S does not contain assignments *derived* at the current decision level. So *Lrn* terminates returning S .

C. D-sequent generation when current target is blocked

Suppose that $Y = \{y\}$ and $F_1 \wedge F_2$ contains (among others) the clauses $C_1 = y \vee x_1$, $C_2 = x_1 \vee \bar{x}_2$, $C_3 = x_2 \vee x_3$. Suppose that C_3 is the current target clause and C_2 is the only clause of $F_1 \wedge F_2$ that can be resolved with C_3 on x_2 .

Suppose *PrvRed* made the assignment $y = 0$. By running *BCP*, *PrvRed* derives the assignment $x_1 = 1$ from clause C_1 . This assignment satisfies C_2 , which makes the target clause C_3 blocked at x_2 . At this point, *Lrn* generates a D-sequent as follows. First, an atomic D-sequent S of the third kind is generated (see Subsection IX-C). S is equal to $(\vec{q}, H) \rightarrow C_3$ where $\vec{q} = (x_1 = 1)$, $H = \emptyset$.

The conditional \vec{q} of S contains the assignment $x_1 = 1$ derived at the current decision level. To get rid of it, *Lrn* joins S with the D-sequent $S' = (\vec{q}', H') \rightarrow C_3$ at x_1 where $\vec{q}' = (y = 0, x_1 = 0)$, $H' = \{C_1\}$. This D-sequent states redundancy of C_3 in the subspace where C_1 is falsified. After joining S and S' , one obtains a new D-sequent S equal to $(\vec{q}, H) \rightarrow C_3$ where $\vec{q} = (y = 0)$, $H = \{C_1\}$. The conditional of S does not contain assignments derived at the current decision level. So S is the final D-sequent returned by *Lrn*.

D. D-sequent generation when a non-target clause is falsified

Generation of a conflict clause C implies that every literal *Lit* resolved out in the process of obtaining C is falsified by the assignment derived from a *clause* (see Appendix III-A). Suppose that at least one such an assignment is derived from a *D-sequent*. Then *Lrn cannot* derive a clause implied by $F_1 \wedge F_2$ and falsified by the current assignment \vec{a} . Instead, *Lrn* derives a *D-sequent* stating redundancy of the current target clause C_{trg} . In this subsection, we consider the case where the clause C_{fls} falsified by *BCP* (i.e. the starting point of *Lrn*) is different from C_{trg} . The next subsection considers the case where $C_{fls} = C_{trg}$.

Suppose that $Y = \{y\}$ and $F_1 \wedge F_2$ contains (among others) the clauses $C_1 = y \vee \bar{x}_1 \vee x_2$, $C_2 = \bar{x}_1 \vee x_3$, $C_3 = \bar{x}_2 \vee \bar{x}_3$. Suppose that the current target clause is $C_4 \in (F_1 \cup F_2)$. Suppose that the D-sequent S^* equal to $(\vec{q}^*, H^*) \rightarrow C_4$ was derived earlier where $\vec{q}^* = (y = 0, x_1 = 0)$ and $H^* = \emptyset$. Assume that *PrvRed* makes the decision assignment $y = 0$. By running *BCP*, *PrvRed* first derives $x_1 = 1$ from S^* . This is due to the fact that S^* becomes unit under $y = 0$ and $x_1 = 1$ *deactivates* S^* (see Subsection VIII-A). Then *PrvRed* derives $x_2 = 1$ and $x_3 = 1$ from C_1 and C_2 respectively. These two assignments falsify C_3 .

The final D-sequent stating redundancy of C_4 is built by *Lrn* as follows. First, *Lrn* generates the D-sequent S equal to $(\vec{q}, H) \rightarrow C_4$ where $\vec{q} = (x_2 = 1, x_3 = 1)$, $H = \{C_3\}$. It is an atomic D-sequent of the second kind stating redundancy of C_4 in the subspace where C_3 is falsified. The conditional \vec{q} of S contains assignments derived at the current decision level. So, *Lrn* picks the most recent derived assignment of \vec{q} i.e. $x_3 = 1$ and gets rid of it. This is achieved by joining S with the D-sequent S' equal to $(\vec{q}', H') \rightarrow C_4$ at variable x_3 where $\vec{q}' = (x_1 = 1, x_3 = 0)$, $H' = \{C_2\}$. The D-sequent S' states redundancy of C_4 in the subspace where C_2 is falsified. The result of joining S and S' is a new D-sequent S equal to $(\vec{q}, H) \rightarrow C_4$ where $\vec{q} = (x_1 = 1, x_2 = 1)$, $H = \{C_2, C_3\}$.

Lrn again picks the most recent derived assignment of \vec{q} i.e. $x_2 = 1$. Then it joins S with the D-sequent S'' equal to $(\vec{q}'', H'') \rightarrow C_4$ at variable x_2 where $\vec{q}'' = (y = 0, x_1 = 1, x_2 = 0)$, $H'' = \{C_1\}$. The D-sequent S'' states redundancy of C_4 in the subspace where C_1 is falsified. The result of joining S and S'' is a new D-sequent S equal to $(\vec{q}, H) \rightarrow C_4$ where $\vec{q} = (y = 0, x_1 = 1)$, $H = \{C_1, C_2, C_3\}$.

Finally, *Lrn* gets rid of the assignment $x_1 = 1$ derived from the D-sequent S^* above. To this end, *Lrn* joins S with S^* at variable x_1 to produce the D-sequent S equal to $(\vec{q}, H) \rightarrow C_4$ where $\vec{q} = (y = 0)$, $H = \{C_1, C_2, C_3\}$. This is the final D-sequent S returned by *Lrn*.

E. D-sequent generation when a target clause is falsified

In this subsection, we continue the topic of the previous subsection. Here, we consider the case $C_{fls} = C_{trg}$ i.e. the current target clause is falsified by *BCP*. (As in the previous subsection, we assume that at least one assignment that “matters” is derived from a D-sequent rather than a clause.) Then *Lrn* still derives a D-sequent S stating redundancy of C_{trg} but also adds a new clause C . The latter is not a full-fledged conflict clause: it may contain *more than one literal* falsified at the conflict level. *Lrn* has to add C to $F_1 \wedge F_2$ to make C_{trg} redundant. *Lrn* derives S and C as follows. The clause C is built similarly to a conflict clause until *Lrn* reaches an assignment derived from a D-sequent. Then *Lrn* builds S by the procedure described in the previous subsection where C is used as a “starting clause” falsified by \vec{a} .

Let us re-examine the example of the previous subsection under the assumption that the falsified clause C_3 is also the current target clause. *Lrn* resolves C_3 with C_2 (on variable x_3) and C_1 (on variable x_2) to produce the clause $C = y \vee \bar{x}_1$. This clause is falsified by the current assignment \vec{a} . Then *Lrn* builds the D-sequent S equal to $(\vec{q}, H) \rightarrow C_3$ where $\vec{q} = (y = 0, x_1 = 1)$ and $H = \{C\}$. This D-sequent states the redundancy of C_3 in the subspace where the new clause C is falsified. Finally, *Lrn* gets rid of $x_1 = 1$ (derived from the D-sequent S^*) in the conditional \vec{q} of S . To this end, *Lrn* joins S with S^* at variable x_1 to produce a new D-sequent S equal to $(\vec{q}, H) \rightarrow C_3$ where $\vec{q} = (y = 0)$ and $H = \{C\}$. Then *Lrn* terminates returning the D-sequent S and clause C .

F. D-sequent generation when a D-sequent is activated

In this subsection, we discuss the case where a D-sequent S derived earlier becomes active. This means that C_{trg} is redundant in the current subspace \vec{a} . If the conditional of S contains assignments derived at the current decision level, Lrn generates a new D-sequent whose conditional does not contain such assignments. Consider the following example. Let $Y = \{y\}$ and $F_1 \wedge F_2$ contain (among others) the clauses $C_1 = y \vee x_1$ and $C_2 = y \vee x_2$. Suppose that clause C_3 of $F_1 \wedge F_2$ is the current target clause. Suppose that the D-sequent S equal to $(\vec{q}, H) \rightarrow C_3$ was derived earlier where $\vec{q} = (x_1 = 1, x_2 = 1)$ and $H = \emptyset$.

Assume that $PrvRed$ made the decision assignment $y = 0$. After running BCP , the assignments $x_1 = 1$ and $x_2 = 1$ are derived from C_1 and C_2 respectively, which activates the D-sequent S . Note that the conditional \vec{q} of S contains assignments derived at the current level. So Lrn generates a new D-sequent as follows. First S is joined with the D-sequent S' equal to $(\vec{q}', H') \rightarrow C_3$ at variable x_2 where $\vec{q}' = (y = 0, x_2 = 0)$, $H' = \{C_2\}$. This D-sequent states the redundancy of C_3 in the subspace where C_2 is falsified. The resulting D-sequent S is equal to $(\vec{q}, H) \rightarrow C_3$ where $\vec{q} = (y = 0, x_1 = 1)$ and $H = \{C_2\}$.

Then S is joined with the D-sequent S'' equal to $(\vec{q}'', H'') \rightarrow C_3$ at variable x_1 where $\vec{q}'' = (y = 0, x_1 = 0)$, $H'' = \{C_1\}$. This D-sequent states the redundancy of C_3 in the subspace where C_1 is falsified. The resulting D-sequent S is equal to $(\vec{q}, H) \rightarrow C_3$ where $\vec{q} = (y = 0)$ and $H = \{C_1, C_2\}$. The conditional of S does not contain assignments derived at the current decision level. So S is the final D-sequent returned by Lrn .

APPENDIX IV

UPDATING STACK OF TARGET LEVELS

In this appendix, we give an example of how the stack T of target levels is updated. Let $\exists X[F(X, Y)]$ be an \exists CNF formula where $F = C_1 \wedge \dots \wedge C_5$. Here $C_1 = y \vee x_1$, $C_2 = \bar{x}_1 \vee x_2$, $C_3 = \bar{x}_2 \vee x_3 \vee x_4$, $C_4 = \bar{y} \vee \bar{x}_3$, $C_5 = x_3 \vee \bar{x}_4$ and $Y = \{y\}$. Consider the PQE problem of taking C_1 out of the scope of quantifiers. So C_1 is the primary target clause and, originally, T has only one target level consisting of C_1 .

Suppose that $PrvRed$ makes the assignment $y = 0$. The BCP procedure finds out that C_1 became a unit clause and adds $x_1 = 1$ derived from C_1 to the assignment queue Q . Since Q does not have any other assignments to make and C_1 is the current target clause C_{trg} , the BCP^* procedure is called (line 4 of Fig. 3). It makes the assignment $x_1 = 1$ derived from C_1 and creates a new top level of T . This level is specified by the pair (C_1, x_1) where C_1 is the key clause and x_1 is the key variable of this level. The latter consists of C_2 , the only clause of F resolvable with C_1 on x_1 . Then BCP^* derives $x_2 = 1$ from C_2 and creates a new top level of T specified by the pair (C_2, x_2) . This level consists of C_3 , the only clause of F resolvable with C_2 on x_2 .

At this point, BCP^* runs out of unit clauses and returns to BCP . Then BCP uses the top level of T to pick the next

target clause T . Since the top level of T consists only of C_3 , the latter is chosen as C_{trg} (line 5) and BCP breaks the while loop (line 8).

APPENDIX V

SPECIAL BACKTRACKING

In this appendix, we discuss backtracking performed by $PrvRed$ when T contains secondary targets. Let us continue considering the example of Appendix IV. After picking C_3 as the current target clause, BCP finds out that C_3 is blocked (line 18 of Fig. 3) at variable x_3 . Indeed, C_4 is the only clause with \bar{x}_3 and it is satisfied by $y = 0$. So BCP returns the backtracking condition $BlkTrg$ (line 19). Then $PrvRed$ learns an atomic D-sequent S of the third kind (line 11 of Fig. 2) equal to $(\vec{q}, H) \rightarrow C_3$ where $\vec{q} = (y = 0)$, $H = \emptyset$.

Since C_3 is a secondary target, $PrvRed$ skips the regular backtracking part (lines 14-17) and goes to the third part of the while loop (lines 18-25). Recall that the current assignment \vec{a} is $(y = 0, x_1 = 1, x_2 = 1)$ and T consists of three target levels. The bottom level of T consists of the primary target C_1 . Then next level is specified by the pair (C_1, x_1) and the top level of T is specified by the pair (C_2, x_2) .

At this point, $PrvRed$ calls the special backtracking procedure $SpecBcktr$ (line 18). Although the conditional of S contains only assignment to y , $SpecBcktr$ cannot undo assignments $x_1 = 1$ and $x_2 = 1$ for the reason explained in Subsection XII-C. The clause C_3 (whose redundancy is stated by S) became a secondary target *only* after the assignment $x_2 = 1$ was made. Since there is no conflict, $SpecBcktr$ cannot backtrack past $x_2 = 1$ (i.e. the “point of origin”). So, $SpecBcktr$ terminates without changing \vec{a} .

Since the conditional of S does not contain any assignments made after $x_2 = 1$, the redundancy of C_3 is proved up to the point of origin. So $PrvRed$ calls $NewTrg$ to pick a new target among the clauses of the top level of T (line 22). As we mentioned earlier, the current top level of T consists only of C_3 . So no new target can be found. As we mentioned in Subsection XII-C, this means that C_2 , the key clause of the top level of T , is blocked at variable x_2 . (Because the only clause of F resolvable with C_2 on x_2 is proved redundant.) So, $NewTrg$ generates an atomic D-sequent S' of the third kind stating the redundancy of C_2 . This D-sequent is equal to $(\vec{q}', H') \rightarrow C_2$ where $\vec{q}' = (y = 0)$ and $H' = \emptyset$. Then $NewTrg$ eliminates the current top level of T restoring the clause C_3 back into the formula F (i.e. treating it as present in formula F). Besides, $NewTrg$ unassigns x_2 .

Now, $NewTrg$ tries to find a target clause in the *new* top level of T specified by the pair (C_1, x_1) . Since the only clause of F resolvable with C_1 on x_1 (i.e. C_2) is proved redundant in the subspace \vec{a} , $NewTrg$ repeats the actions described before. First, it derives an atomic D-sequent S'' of the third kind stating the redundancy of C_1 . Here S'' is equal to $(\vec{q}'', H'') \rightarrow C_1$ where $\vec{q}'' = (y = 0)$ and $H'' = \emptyset$. Then it eliminates the top level of T restoring C_2 back into formula F and unassigning x_1 .

Now, T is reduced to the primary target level (containing clause C_1). $NewTrg$ terminates returning C_1 as C_{trg} and S'' as

D-sequent S (line 22). Since C_1 is the primary target, $PrvRed$ checks if the conditional of S is empty (line 24). Since it is not, the redundancy of C_1 is proved only in the subspace \vec{a} that is currently equal to $(y = 0)$. Since S is unit in the subspace \vec{a} , $PrvRed$ adds the assignment $y = 1$ derived from S to the assignment queue Q (line 25). Then $PrvRed$ starts a new iteration of the while loop.

APPENDIX VI CORRECTNESS OF $DS-PQE^+$

In this appendix, we give an informal proof that $DS-PQE^+$ is sound and complete.

A. $DS-PQE^+$ is sound

Let $\exists X[F_1(X, Y) \wedge F_2(X, Y)]$ be an \exists CNF formula. Suppose that $DS-PQE^+$ is used to take F_1 out of the scope of quantifiers. Assume, for the sake of simplicity, that every clause of F_1 is an X -clause. In its operation, $DS-PQE^+$ generates new clauses obtained by resolving clauses of $F_1 \wedge F_2$ and thus implied by $F_1 \wedge F_2$. The final formula produced by $DS-PQE^+$ can be represented as $\exists X[F]$ where $F = F_1^{ini} \wedge F_1^* \wedge F_1^{**} \wedge F_2^{ini} \wedge F_2^*$. Here F_1^{ini} and F_2^{ini} denote the initial versions of F_1 and F_2 respectively. The formula $F_1^*(Y)$ denotes the derived clauses depending only on variables of Y whose generation involved clauses of F_1^{ini} and/or their descendants. The formula $F_1^{**}(X, Y)$ denotes the derived X -clauses whose generation involved clauses of F_1^{ini} and/or their descendants. The formula $F_2^*(X, Y)$ denotes the derived clauses whose generation involved *only* clauses of F_2^{ini} and/or their descendants.

For every clause C of $F_1^{ini} \wedge F_1^{**}$, $DS-PQE^+$ calls $PrvRed$ that generates a D-sequent S equal to $(\vec{q}, H) \rightarrow C$ where $\vec{q} = \emptyset$ and $H \subseteq (F \setminus \{C\})$. The D-sequent S states redundancy of C in formula $\exists X[F]$. This D-sequent is correct due to correctness of the atomic D-sequents and the join operation and due to Proposition 5 of Appendix II. The D-sequents for the clauses of $F_1^{ini} \wedge F_1^{**}$ are derived by $DS-PQE^+$ one by one in some order. That is these D-sequents are *consistent* (see Definition 13). So, one can claim that $\exists X[F_1^{ini} \wedge F_2^{ini}] \equiv \exists X[F] \equiv F_1^* \wedge \exists X[F_2^{ini} \wedge F_2^*] \equiv F_1^* \wedge \exists X[F_2^{ini}]$. Thus, $F_1^*(Y)$ is indeed a solution to the PQE problem at hand.

B. $DS-PQE^+$ is complete

Assume that $DS-PQE^+$ generates only clauses that have not been seen before. Taking into account that $DS-PQE^+$ examines a finite search tree, this means that $DS-PQE^+$ always terminates and thus is complete. The problem however is that the version of $PrvRed$ described in Section XI *may* generate a duplicate of an X -clause that is currently *proved redundant*. So $PrvRed$ and hence $DS-PQE^+$ may loop.

To prevent looping, the current implementation of $PrvRed$ does the following. (For the sake of simplicity, we did not discuss this part of $PrvRed$ in Section XI.) Let (\vec{y}, \vec{x}) be the current assignment to $Y \cup X$ made by $PrvRed$ before a duplicate of an X -clause is generated. After a duplicate C is generated, $PrvRed$ discards C and backtracks to the

last assignment to a variable of Y (and thus undoing all assignments to X). This is accompanied by removing all secondary targets from the stack T . So, on completion of backtracking, the primary target clause C_{pr} is the current target clause. Then $PrvRed$ generates a D-sequent stating the redundancy of C_{pr} in subspace \vec{y} and keeps going as if $PrvRed$ just finished line 11 of Figure 2.

To generate the D-sequent above, $PrvRed$ does the following. First, $PrvRed$ runs an internal SAT-solver to check if formula F (defined in the previous subsection) is satisfiable in subspace \vec{y} . If not, a clause $C(Y)$ implied by F is generated and added to F . Then an atomic D-sequent of the second kind is generated stating the redundancy of C_{pr} in the subspace where $C(Y)$ is falsified. Otherwise, $PrvRed$ finds an assignment (\vec{y}, \vec{x}) satisfying F . The existence of such an assignment means that C_{pr} is redundant in the subspace \vec{y} without adding any clauses. Then $PrvRed$ generates a D-sequent $(\vec{y}^*, H) \rightarrow C_{pr}$ where $H = \emptyset$ and $\vec{y}^* \subseteq \vec{y}$ and (\vec{y}^*, \vec{x}) satisfies F . (In other words, $PrvRed$ tries to shorten the satisfying assignment to reduce the conditional of the D-sequent constructed for C_{pr})

APPENDIX VII REDUCING SIZE OF STRUCTURE CONSTRAINTS

In this appendix, we describe some methods for reducing the size of structure constraints in D-sequents learned by $DS-PQE^+$. Let S be a D-sequent $(\vec{q}, H) \rightarrow C$ stating redundancy of C in $\exists X[F_1(X, Y) \wedge F_2(X, Y)]$. A useful observation here is that one does not need to keep a clause $B \in H$ if $\text{Vars}(B) \subseteq Y$. Indeed, $DS-PQE^+$ proves redundancy only of X -clauses. So a clause $B(Y)$ is either present in H or is satisfied by the current assignment \vec{a} . In either case, S can be safely reused in the subspace \vec{a} (see Proposition 7).

As we mentioned earlier, one can keep structure constraints small by generating only D-sequents for the target clauses of the k bottom levels of the stack T (see Section XIII). In this case, the size of H is limited by the total number of secondary target clauses of levels $1, \dots, k$. In particular, one can safely reuse the D-sequents of the primary target clause (level 0 of T) without computing structure constraints at all¹¹.

Another method is based on using the substitution operation (see Appendix II). By repeatedly applying this operation, one can reduce the structure constraint of a D-sequent S to an empty set (which may increase the conditional of S) [21].

Finally, one can reduce the size of structure constraints by adding new clauses. Consider the following example. Suppose that $\exists X[F_1(X, Y) \wedge F_2(X, Y)]$ contains (among others) clauses $C_1 = y \vee x_1$, $C_2 = \bar{x}_1 \vee x_2$, ..., $C_k = \bar{x}_{k-1} \vee x_k$, $C_{k+1} = x_k \vee x_{k+1}$. Suppose C_{k+1} is the current target clause. Suppose that $Y = \{y\}$ and $PrvRed$ made the assignment

¹¹ Let $(\vec{q}, H) \rightarrow C_{trg}$ be a D-sequent S where C_{trg} is the current target. Suppose $\vec{q} \subseteq \vec{a}$ holds where \vec{a} is the current subspace examined by $DS-PQE^+$. If C_{trg} is the primary target, no X -clause of H is a secondary target (because the set of secondary targets is currently empty). So a clause of H is either present in $F_1 \wedge F_2$ or *satisfied* by \vec{a} . In either case, S can be safely reused in the subspace \vec{a} (see Proposition 7).

$y = 0$. After running BCP, *PrvRed* derives $x_2 = 1$ from C_2 , $x_3 = 1$ from C_3 and so on until $x_k = 1$ is derived from C_k . The latter assignment satisfies the target clause C_{k+1} . In this case, in our current implementation, the *Lrn* procedure generates the D-sequent S equal to $(\vec{q}, H) \rightarrow C_{k+1}$ where $\vec{q} = (y = 0)$ and $H = \{C_1, \dots, C_k\}$. (This D-sequent is constructed as described in Appendix III-B where one performs k join operations at variables x_k, \dots, x_1 .) Note that no *new* clauses are added when building S . One can reduce the size of H by generating a new clause $C = y \vee x_k$ obtained by resolving clauses C_k, \dots, C_1 on variables x_k, \dots, x_1 . Adding C to $F_1 \wedge F_2$ makes C_{k+1} redundant in subspace $y = 0$. So one can derive the D-sequent $(\vec{q}, H) \rightarrow C_{k+1}$ where $\vec{q} = (y = 0)$, $H = \{C\}$ stating redundancy of C_{k+1} in $\exists X[C \wedge F_1 \wedge F_2]$. Thus, one reduces the size of the structure constraint H at the expense of adding a new clause.

APPENDIX VIII ONE MORE EXPERIMENT

In Section XIII, we describe some experiments with $DS-PQE^+$. In this appendix, we describe one more experiment. Similarly to the experiments of Section XIII, $DS-PQE^+$ stored/reused only D-sequents of the primary target clauses. This could not affect the results of Section XIII much because, for the problem we considered there, generation of a secondary target clause was a rare event. This was not true for the problem considered in this appendix where a very large number of secondary targets was generated. Nevertheless, the experimental results presented in this appendix show that reusing even a small fraction of D-sequents can be beneficial.

In this section, we solve the PQE problem arising in the method of equivalence checking introduced in [19]. (The main idea of this method is sketched in Subsection III-D.) Let $N'(X', Y', z')$ and $N''(X'', Y'', z'')$ be single-output circuits to be checked for equivalence. Let T_{cut} specify the variables of a cut in N' and N'' . Let F_{cut} specify the gates of N' and N'' located between the inputs and the cut. Let W_{cut} denote $Vars(F_{cut}) \setminus T_{cut}$. Let $EQ(X', X'')$ be a formula evaluating to 1 iff X' and X'' have the same assignments. We consider the problem of taking EQ out of the scope of quantifiers in $\exists W_{cut}[EQ \wedge F_{cut}]$. That is one needs to find a formula $G(T_{cut})$ such that $G \wedge \exists W_{cut}[F_{cut}] \equiv \exists W_{cut}[EQ \wedge F_{cut}]$.

In this experiment, we used HWMCC-10 benchmarks. Circuit N' was specified by the transition relation of a benchmark and N'' was obtained from N' by a logic optimization tool. A cut of N' and N'' was formed from gates located in N' and N'' at topological levels $\leq k$. (The set consisting only of gates of topological level k , in general, does not form a cut.) The input variables have topological level 0. Table IV shows results for a sample of the set of benchmarks we tried.

The first column gives the name of a benchmark. The second column gives the value of k above. The following columns give the number of generated D-sequents (in thousands) and the run time for the PQE procedures we compared. In this experiment, we compared the same procedures as in Tables I. The results of Table IV show that $DS-PQE^+$ with limited

TABLE IV
COMPUTING CUT CONSTRAINS IN EQUIVALENCE CHECKING. THE TIME LIMIT IS SET TO 100 SECONDS

name	k	$DS-PQE$		$DS-PQE^+$ no learning		$DS-PQE^+$ limited learning	
		#dseqs $\times 10^3$	time (s.)	#dseqs $\times 10^3$	time (s.)	#dseqs $\times 10^3$	time (s.)
kenooppl	5	>22,500	*	4	0.3	2.9	0.2
bj08autg3f1	50	424	0.8	15	0.6	7.2	0.3
abp4pold	2	>18,786	*	25	0.9	7	0.2
eijks838	5	302	0.6	43	1.0	52	1.1
pdtpmstwo	3	1,345	2.1	310	5.1	96	1.9
prodconspld4	15	89	0.2	1.7	0.02	2	0.02
kenoop2	2	>12,985	*	4.4	1.3	3.3	1.1
texastwoprocp1	12	465	0.8	68	1.0	17	0.3
eijks953	3	331	2.0	1	0.06	0.6	0.05
bj08amba2g1	3	2,414	6.0	3.5	0.6	1.7	0.3
pdvisthuffman1	40	1,656	1.8	1	0.3	0.6	0.2
pdtpmsheap	3	2,039	6.0	3.7	0.08	2.1	0.07
pdvistimeout1	16	22,902	39	8.9	0.03	8	0.02
brp2time	2	>16,841	*	9.1	1.0	4.1	0.4

learning outperforms $DS-PQE$ and $DS-PQE^+$ without learning.