Partial Quantifier Elimination By Certificate Clauses

Eugene Goldberg

eu.goldberg@gmail.com

Abstract. We study a modification of the Quantifier Elimination (QE) problem called Partial QE (PQE) for propositional CNF formulas. In PQE, only a small subset of target clauses 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 can be dramatically simpler than QE. One can perform PQE by adding a set of clauses depending only on free variables that make the target clauses redundant. Proving redundancy of a target clause is done by derivation of a "certificate" clause *implying* the former. This idea is implemented in our PQE algorithm called *START*. It bears some similarity to a SAT-solver with conflict driven learning. A major difference here is that *START* backtracks as soon as a target clause is proved redundant (even if no conflict occurred). We experimentally evaluate *START* on a practical problem. We use this problem to show that PQE can be much easier than QE.

1 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 hard: in the worst case, finding $F^*(Y)$ reduces to solving $2^{|Y|}$ satisfiability (SAT) problems.

There are at least two ways of making QE easier to solve. First, one can consider only instances of QE where $|Y|$ is small. In particular, if $|Y| = 0$, QE reduces to a SAT problem. This line of research featuring efficient verification algorithms based on SAT (e.g. [4,34,8]) has gained great popularity. Another way to address the complexity of QE suggested in [24] 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*. We will call F_1^* a *solution* for the PQE problem above.

The appeal of PQE is fourfold. First, PQE can be exponentially more efficient than QE [14]. Second, in many verification problems, one can replace QE with PQE. Third, PQE can be used to generate design properties and so address the problem of incompleteness of specification (see Section 3). Fourth, quantifiers give PQE extra semantic power over pure SAT-solving. For instance, an

equivalence checker based on PQE [18] 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 redundancy[1] in a *quantified* formula and thus cannot be generated by a traditional SAT-solver. In [19], it is shown that a PQE-solver can compute the reachability diameter and so can turn bounded model checking [4] into unbounded one (as opposed to a pure SAT-solver).

A solution $F_1^*(Y)$ to the PQE problem can be found using the two observations below. First, $(F_1 \wedge F_2) \Rightarrow F_1^*$ (i.e. $F_1 \wedge F_2$ implies F_1^*). So F_1^* can be obtained by resolving clauses of $F_1 \wedge F_2$. Second, $F_1^* \wedge \exists X[F_1 \wedge F_2] \equiv F_1^* \wedge \exists X[F_2]$. Thus, a formula F_1^* implied by $F_1 \wedge F_2$ becomes a solution as soon as F_1^* makes the clauses of F_1 *redundant*. Note that resolution alone is not sufficient to express redundancy in a quantified formula. Suppose, for instance, that a clause C is redundant in $\exists X[G(X) \wedge C]$. In general, $G \not\Rightarrow C$. So, one cannot certify redundancy of C by resolving clauses of G to derive a clause K implying C. In [22,20], we addressed the problem above by the machinery of D-sequents. A D-sequent is a record claiming redundancy of a clause in a specified subspace. The machinery of D-sequents has two flaws. First, the semantics of a D-sequent is complicated, which makes it hard to prove the correctness of an algorithm employing D-sequents. Second, to reuse a D-sequent in different branches of the search tree, one has to keep a lot of contextual information. This makes D-sequent reusing quite expensive.

The contribution of this paper is as follows. *First*, we show how the problem above can be solved by combining resolution with adding clauses that preserve equisatisfiability rather than equivalence. Then, in the example above, one can always derive a clause K implying clause C to certify that the latter is redundant in $\exists X[G \wedge C]$. The clause K is called a *certificate* clause. Importantly, the semantics of certificates is easy and reusing certificate clauses does not require storing any contextual information (as opposed to D-sequents). *Second*, we implement our approach in a PQE algorithm called *START*, an abbreviation of Single TARgeT. At any given moment, *START* proves redundancy[2] of only one clause (hence the name "single target"). *START* is somewhat similar to a SAT-solver with conflict driven learning. A major difference here is that *START* backtracks as soon as a target clause is proved redundant in the current subspace (even if no conflict occurred). *Third*, we provide an experimental evaluation of *START* on the problem of computing range reduction (in the context of testing). Range reduction can also be computed by QE. We use *START* and high-quality QE tools like CADET [35] to show that PQE can be dramatically simpler than QE.

[1] A *clause* is a disjunction of literals (where a literal of a Boolean variable w is either w itself or its negation $\overline{w}$). So a CNF formula F is a conjunction of clauses: $C_1 \wedge \cdots \wedge C_k$. We also consider F as the *set of clauses* $\{C_1, \ldots, C_k\}$. Clause C is redundant in $\exists X[F]$ if $\exists X[F] \equiv \exists X[F \setminus \{C\}]$.

[2] By "proving a clause C redundant" we mean showing that C is redundant after adding some new clauses (if necessary).

The main body[3] of this paper is structured as follows. Sections 2-3 discuss the appeal of PQE. Basic definitions are provided in Sections 4-5. In Section 6, we give an example of the PQE problem. PQE by computing redundancy is introduced in Section 7. *START* is described in Sections 8-10. Section 11 provides experimental results. Some background is given in Section 12. In Section 13, we make conclusions.

2 The Appeal Of PQE-I

We study PQE by a *top-down* approach. We start with arguing the importance of PQE. Then we work on an efficient PQE algorithm. (A more traditional approach is to build tools around an *existing* efficient technique e.g. SAT-solving.) The appeal of PQE is twofold. First, many verification problems reduce to PQE. Some of the them are listed in this section and section 11. Second, PQE can be used to generate design properties and thus address the problem of incompleteness of specification (see the next section).

2.1 SAT-solving by PQE [24]

Let $F(X)$ be a formula to check for satisfiability and $\vec{x}$ be a full assignment to X. Let F_1 and F_2 denote the clauses of F falsified and satisfied by $\vec{x}$ respectively. Checking the satisfiability of F reduces to finding a Boolean constant F_1^* such that $F_1^* \wedge \exists X[F_2] \equiv \exists X[F_1 \wedge F_2]$ (see [24]). F_1^* is a constant because all variables of F are quantified. If $F_1^* = 0$, then F is unsatisfiable. Otherwise, i.e. if $F_1^* = 1$, formula F is satisfiable (since $\vec{x}$ satisfies F_2).

Note that SAT can be viewed as QE for the formula $\exists X[F(X)]$. Obviously, it is a *special case* of QE since all variables are quantified. As in every QE problem, one needs to prove *all* clauses of F redundant e.g. by deriving an empty clause or finding a satisfying assignment. So formulating SAT as PQE is an improvement since it suffices to prove redundancy of a *subset* of clauses.

2.2 Interpolation by PQE [17]

Let $A(X,Y) \wedge B(Y,Z)$ be a formula where X, Y, Z are sets of variables. Let $A^*(Y)$ be a formula such that $\exists W[A \wedge B] \equiv A^* \wedge \exists W[B]$ where $W = X \cup Z$. Assume $A \Rightarrow A^*$ and $A \wedge B \equiv 0$. Then A^* is an interpolant [12,34]. So, one can view interpolation as a special case of PQE [17].

The benefit of formulating interpolation in terms of PQE is twofold. Suppose that $A \wedge B \equiv 0$ but $A \not\Rightarrow A^*$. Then one can build an interpolant off the resolution derivation of A^* like it is done in [34] where an interpolant is built off a proof that $A \wedge B$ is unsatisfiable. This makes sense because derivation of A^* can be much simpler than this proof. Another benefit of using PQE is that it allows to extend the notion an interpolant to the case where $A \wedge B$ is *satisfiable*. The idea here is that an assignment satisfying $A^* \wedge B$ can be viewed as an "abstract counterexample" for $A \wedge B \equiv 0$.

[3] Some additional information is provided in appendices.

2.3 Equivalence checking by PQE [18]

Let $N_1(X_1, Y_1, z_1)$ and $N_2(X_2, Y_2, z_2)$ be single-output combinational circuits to check for equivalence. Here X_i, Y_i are sets of input and internal variables and z_i is the output variable of $N_i, i = 1, 2$. Let $EQ(X_1, X_2)$ specify the predicate such that $EQ(\vec{x}_1, \vec{x}_2) = 1$ iff $\vec{x}_1 = \vec{x}_2$. Here $\vec{x}_i$ is a full assignment to X_i, $i = 1, 2$. Let formulas $G_1(X_1, Y_1, z_1)$ and $G_2(X_2, Y_2, z_2)$ specify[4] circuits N_1 and N_2 respectively. Let $h(z_1, z_2)$ be a formula such that $\exists W[EQ \wedge G_1 \wedge G_2] \equiv h \wedge \exists W[G_1 \wedge G_2]$ where $W = X_1 \cup Y_1 \cup X_2 \cup Y_2$. If h specifies $z_1 \equiv z_2$, then N_1 and N_2 are equivalent. Otherwise, they are inequivalent[5] [18].

Using PQE for equivalence checking allows for exploiting the structural similarity of N_1 and N_2 to the fullest extent. In particular, the PQE-based algorithm of [18] benefits from existence of *any* short relationships between internal points of N_1 and N_2. On the other hand, the current tools only search for *some predefined relations* like equivalence between internal points of N_1 and N_2.

2.4 Computing Reachability Diameter by PQE [19]

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 [19]. Let $I_0 = I(S_0)$ and $I_1 = I(S_1)$ and $W_n = S_0 \cup \cdots \cup S_n$ and $T_{i,i+1} = T(S_i, S_{i+1})$ and $G_{0,n} = T_{0,1} \wedge \cdots \wedge T_{n-1,n}$. Testing if $n \geq Diam(I, T)$ reduces to checking if $\exists W_n[I_1 \wedge I_0 \wedge G_{0,n+1}] \equiv \exists W_n[I_0 \wedge G_{0,n+1}]$ i.e. whether I_1 is redundant. If so, then $n \geq Diam(I, T)$ and to prove a safety property, it suffices to run bounded model checking [4]. (Namely, it suffices to show that no counterexample of length n or less exists.)

An obvious advantage of using PQE to compute the reachability diameter is that one *does not need* to generate the set of *all* reachable states. This is an example of a problem that cannot be solved by a SAT-solver and that is hard for an algorithm performing *complete* QE.

3 The Appeal Of PQE-II

In the previous section, we listed some verification problems that reduce to PQE. The appeal of PQE is also that it can be used to generate design properties and

[4] Let $N(X, Y, Z)$ be a combinational circuit where X, Y, Z specify the sets of input, internal and output variables of N respectively. A formula $F(X, Y, Z)$ **specifies** the circuit N, if every assignment satisfying F corresponds to a consistent assignment to the variables of N and vice versa. An assignment to the variables of N is called consistent if it is consistent for every gate G of N i.e. the value assigned to the output variable of G is implied by the values assigned to the input variables of G.

[5] There is one very rare exception here (see [18]). The fact that h does not specify $z_1 \equiv z_2$ may also mean that both N_1 and N_2 implement the same Boolean constant and hence are equivalent. This possibility can be checked by a few easy SAT calls.

thus address the problem of *incompleteness of specification* [14,21]. Consider this issue by the design of a combinational circuit. Let $T_{tbl}(X, Z)$ denote the "truth table property" of a correct implementation where X and Z are sets of input and output variables respectively. (That is input $\vec{x}$ produces output $\vec{z}$ iff $T_{tbl}(\vec{x}, \vec{z}) = 1$.) Let circuit $N(X, Y, Z)$ implement T_{tbl} where Y is the set of internal variables. Let $F(X, Y, Z)$ be a formula describing[6] the functionality of N. Then verifying N reduces to checking if $T_{tbl}(X, Z) \equiv \exists Y[F]$.

Now suppose N is defined not by T_{tbl} but via weaker properties $P_1, \ldots, P_k$ forming a specification $\mathbb{P}$. (If $P_i(\vec{x}, \vec{z}) = 0$, input $\vec{x}$ cannot produce output $\vec{z}$.) If $F \Rightarrow P_i$, $i = 1, \ldots, k$, the circuit N is considered to be correct. Unfortunately, $\mathbb{P}$ is typically incomplete i.e. $P_1 \wedge \ldots \wedge P_k \not\Rightarrow \exists Y[F]$. This leads to two problems. First, N may have some *unwanted* properties allowed by $\mathbb{P}$. Second, N may break some *desired* properties of N absent from $\mathbb{P}$. In either case, N is buggy.

PQE can address the incompleteness of $\mathbb{P}$. Let $Q(X, Z)$ be a formula obtained by taking a clause $C \in F$ out of the scope of quantifiers in $\exists Y[C \wedge F']$ where $F' = F \setminus \{C\}$. That is $\exists Y[C \wedge F'] \equiv Q \wedge \exists Y[F']$. Since Q is implied by F, it is a *property* of N. If $P_1 \wedge \cdots \wedge P_k \not\Rightarrow Q$, then Q identified a hole in specification. If Q is an unwanted property i.e. Q forbids some *correct* input/output behaviors, N is buggy. (An example of an unwanted property is given in Appendix A.) Otherwise, one needs to fix the hole in $\mathbb{P}$ e.g by adding Q to $\mathbb{P}$. By taking different clauses of F out of the scope of quantifiers and updating $\mathbb{P}$ one makes the latter "structurally complete". That is properties of $\mathbb{P}$ probe every piece of design. (This idea comes from testing where the design *coverage* by a test set makes up for the functional incompleteness of the latter and where this compromise works quite well.)

Summarizing, one can say that QE builds the *strongest property* (e.g. $\exists Y[F]$) whereas PQE produces *weaker* ones. The idea of using PQE for property generation can be extended in three directions. First, one can reduce the complexity of PQE (and so the strength of property Q) as follows. The clause C is replaced with $m+1$ clauses $C \vee l(v_1), \ldots, C \vee l(v_m)$, $C \vee \overline{l(v_1)} \vee \cdots \vee \overline{l(v_m)}$ and the latter clause is taken out of the scope of quantifiers instead of C. Here $v_i \in (X \cup Y \cup Z)$ and $l(v_i)$ is a literal of v_i, $i = 1, \ldots, m$ In fact, the complexity of PQE can be reduced to *linear* [14] (which shows that PQE can be *exponentially* more efficient than QE). Second, PQE can be used to generate *false* properties of N. They are supposed to imitate the desired properties not met by N due to their absence from $\mathbb{P}$. Tests breaking these properties generated by PQE [21] can identify bugs caused by incompleteness of $\mathbb{P}$ (if any). Third, one can move on to *sequential* circuits and use PQE to generate true and false *safety* properties [14,21].

4 Basic Definitions

In this paper, we consider only propositional CNF formulas. In this section, when we say "formula" without mentioning quantifiers, we mean a *quantifier-free* formula.

[6] As usual, F is built by conjoining formulas specifying the gates of N.

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

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

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 refer to $\vec{q}$ as a* ***full assignment*** *to V if $V' = V$. We will denote the set of variables assigned in $\vec{q}$ as* $\boldsymbol{Vars(\vec{q})}$*. We will denote as* $\boldsymbol{\vec{q} \subseteq \vec{r}}$ *the fact that a) $Vars(\vec{q}) \subseteq Vars(\vec{r})$ and b) every variable of $Vars(\vec{q})$ 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.* $\boldsymbol{C_{\vec{q}} \equiv 1}$ *if C is satisfied by $\vec{q}$. Otherwise,* $\boldsymbol{C_{\vec{q}}}$ *is the clause obtained from C by removing all literals falsified by $\vec{q}$.* $\boldsymbol{H_{\vec{q}}}$ *denotes the formula obtained from H by removing the clauses satisfied by $\vec{q}$ and replacing every clause C unsatisfied by $\vec{q}$ with $C_{\vec{q}}$.*

Definition 5. *Let $\exists X[F(X,Y)]$ be an ∃CNF formula. A clause C of F is called an* ***X-clause*** *if $Vars(C) \cap X \neq \emptyset$.*

Definition 6. *Let G, H be formulas that may have quantifiers. We say that G, H are* ***equivalent****, written* $\boldsymbol{G \equiv H}$*, if for all assignments $\vec{q}$ where $Vars(\vec{q}) \supseteq (Vars(G) \cup Vars(H))$, we have $G_{\vec{q}} = H_{\vec{q}}$.*

Definition 7. *Let F be a 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*** $\boldsymbol{\exists X[F]}$ *if $\exists X[F] \equiv \exists X[F \setminus G]$.*

Definition 8. *The* ***Quantifier Elimination (QE)*** *problem specified by $\exists X[F(X,Y)]$ is to find formula $F^*(Y)$ such that* $\boldsymbol{F^* \equiv \exists X[F]}$*.*

Definition 9. *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* $\boldsymbol{\exists X[F_1 \wedge F_2] \equiv F_1^* \wedge \exists X[F_2]}$*. Formula F_1^* is called a* ***solution*** *to PQE.*

Remark 1. Note that if F_1^* is a solution to the PQE problem above and a clause $C \in F_1^*$ is implied by F_2 *alone*, then $F_1^* \setminus \{C\}$ is a solution too. So F_1 is *not redundant* in $\exists X[F_1 \wedge F_2]$ *only* if at least one clause of F_1^* is not implied by F_2.

5 Implication By Equisatisfiability

The relation of implication can be introduced via that of equivalence. Namely, formula G is implied by formula F iff $F \wedge G \equiv F$. This observation can be used for introducing implication with respect to equisatisfiability. This type of implication is very useful in describing PQE by computing redundancy.

Definition 10. *Let $F(X,Y)$ and $G(X,Y)$ be formulas and $\exists X[F \wedge G] \equiv \exists X[F]$ (and so $(F \wedge G)_{\vec{y}}$ and $F_{\vec{y}}$ are equisatisfiable for every full assignment $\vec{y}$ to Y). We will say that G is* ***es-implied*** *by F* ***with respect to Y****. A clause C is called an* ***es-clause*** *with respect to F and Y if it is es-implied by F with respect to Y.*

We will use symbols $\Rightarrow$ and $\ddot{\Rightarrow}$ to denote regular implication and es-implication respectively. Note that if $F \Rightarrow G$, then $F \ddot{\Rightarrow} G$ with respect to Y. However, the opposite is not true. We will just say that F es-implies G without mentioning Y if the latter is clear from the context or empty.

Definition 11. *Let clauses C',C'' have opposite literals of exactly one variable $w \in Vars(C') \cap Vars(C'')$. Then clauses C',C'' are called* ***resolvable*** *on w. The clause C having all literals of C', C'' but those of w is called the* ***resolvent*** *of C',C'' on w. The clause C is said to be obtained by* ***resolution*** *on w.*

Definition 12. *Let $F(X,Y)$ be a formula and $C(X,Y)$ be a clause. Let G be the set of clauses of F resolvable with C on a variable $w \in X$. Let $w = b$ where $b \in \{0,1\}$ satisfy C. The clause C is called* ***blocked*** *in F at w with respect to Y if $(F \setminus G)_{w=b} \ddot{\Rightarrow} G_{w=b}$ with respect to Y. (In particular, C is blocked if G is an empty set.) This definition modifies that of a blocked clause given in [31,16].*

Proposition 1. *Let $F(X,Y)$ be a formula and $C(X,Y)$ be a clause blocked in F at $w \in X$ with respect to Y. Then $F \ddot{\Rightarrow} C$ with respect to Y.*

Proofs of the propositions are given in Appendix B.

6 A Simple Example Of PQE

In this section, we present an example of PQE by computing clause redundancy. Let $\exists X[C_1 \wedge F]$ be an $\exists$CNF where $X = \{x_1, x_2\}$, $C_1 = \overline{x}_1 \vee x_2$, $F = C_2 \wedge C_3$, $C_2 = y \vee x_1$, $C_3 = y \vee \overline{x}_2$. The set Y of unquantified variables is $\{y\}$. Below, we take C_1 out of the scope of quantifiers by proving it 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. By conflict analysis [32], one derives the conflict clause $C_4 = y$ (obtained by resolving C_1 with clauses C_2 and C_3). Adding C_4 to $C_1 \wedge F$ *makes* C_1 redundant in subspace $y = 0$. Note that C_1 is not redundant in $\exists X[C_1 \wedge F]$ in subspace $y = 0$. Formula F is satisfiable in subspace $y = 0$ whereas $C_1 \wedge F$ is not. So, one *has to* add C_4 to make C_1 redundant in this subspace.

In subspace $y = 1$, C_1 is blocked at x_1. (C_1 is resolvable on x_1 only with C_2 that is satisfied by $y = 1$ and hence is redundant in subspace $y = 1$.) So C_1 is redundant in formula $\exists X[C_4 \wedge F]$ when $y = 1$. This redundancy can be certified by the clause $C_5 = \overline{y} \vee \overline{x}_1$ that *implies* C_1 in subspace $y = 1$. Note that C_5 is blocked in formula $C_1 \wedge C_4 \wedge F$ at x_1 (because C_5 and C_2 are unresolvable on x_1). So, from Proposition 1 it follows that this formula es-implies C_5 with respect to $\{y\}$. (The construction of clauses like C_5 is described in Subsection 8.4.) Adding C_5 is *optional* because C_1 is already redundant in subspace $y = 1$.

By resolving clauses C_4 and C_5 one derives the clause $C_6 = \overline{x}_1$ that implies C_1. The clause C_6 serves just as a certificate of the global redundancy of C_1. Thus, like C_5, it does not have to be added to the formula. So, $\exists X[C_1 \wedge F \wedge C_4] \equiv C_4 \wedge \exists X[F]$. Since $C_1 \wedge F$ implies C_4, then $\exists X[C_1 \wedge F] \equiv C_4 \wedge \exists X[F]$. So C_4 is a solution to our PQE problem.

7 PQE By Proving Redundancy

In this section, we generalize the example of Section 6 where redundancy is proved by branching. This generalization also sketches the main idea of *START*, a PQE algorithm described later. Consider the following PQE problem. Given a formula $F(X,Y)$ and a clause C_{trg}, find formula $H(Y)$ such that $H \wedge \exists X[F] \equiv \exists X[C_{trg} \wedge F]$ (where "trg" stands for "target"). By assigning variables of $X \cup Y$, one eventually reaches a subspace $\vec{a}$ where a clause implying C_{trg} in this subspace can be easily derived (which proves C_{trg} redundant). One can derive two kinds of clauses implying C_{trg} in subspace $\vec{a}$. (The clauses of both kinds are implied or es-implied by the current formula F.) A clause of the first kind is derived using C_{trg}. Such a clause *has to* be added to the formula to guarantee that C_{trg} is redundant in subspace $\vec{a}$. A clause of the second kind is derived without using C_{trg}. It just serves as a witness of the redundancy of C_{trg}. So, adding such a clause to the formula is *optional*.

By resolving clauses implying C_{trg} in different branches one eventually derives a clause K such that $K \Rightarrow C_{trg}$ and $F \ddot{\Rightarrow} K$. Then $\exists X[C_{trg} \wedge F] \equiv \exists X[F]$. Let H denote the clauses to F that were obtained using C_{trg} (i.e. clauses of the first kind). Let $F' = F \setminus H$. Since $(C_{trg} \wedge F') \ddot{\Rightarrow} H$, then $\exists X[C_{trg} \wedge F'] \equiv \exists X[H \wedge F']$. Let F^{ini} denote the initial version of formula F. Formula F' is obtained from F^{ini} by adding clauses of the second kind. These clauses are es-implied by F^{ini} with and without C_{trg} (see Subsection 8.4). Thus, F' can be replaced with F^{ini} and so $\exists X[C_{trg} \wedge F^{ini}] \equiv \exists X[H \wedge F^{ini}]$. If all clauses of H depend only on Y, H is a solution to the PQE problem at hand. Otherwise, one keeps applying the algorithm above to the X-clauses of H (one by one) until H depends only on Y.

8 Introducing *START*

In this section, we give a high-level view of the PQE algorithm called *START* (an abbreviation of Single TARgeT). A more detailed description of *START* is presented in the next two sections. A proof of correctness of *START* is given in Appendix C.

8.1 Main loop of *START*

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

```
// λ denotes (C_trg,X,a⃗)
//
START(F_1, F_2 ‖ X){
1 while (true) {
2   C_trg := PickXcls(F_1)
3   if (C_trg = nil) {
4     F_1^* = F_1
5     return(F_1^*)}
6   a⃗ := ∅
7   K := PrvRed(F_1,F_2‖λ)
8   if (Empty(K)) return(K)
9   F_1 := F_1 \ {C_trg}  }}
```

Fig. 1: Main loop

START runs a while loop that begins with picking an X-clause $C_{trg} \in F_1$ (line 2). We will refer to C_{trg} as the **target clause**. If F_1 has no X-clauses, it is the solution $F_1^*(Y)$ returned by *START* (lines 3-5). Otherwise, *START* sets the initial assignment $\vec{a}$ to $X \cup Y$ to $\emptyset$ and invokes a procedure called *PrvRed* to prove C_{trg} redundant (line 7). The latter may require adding new clauses to F_1 and F_2. In particular, *PrvRed* may add to F_1 new X-clauses to prove redundant in later iterations of the loop. *PrvRed* returns a clause K certifying redundancy of C_{trg} (see Subsection 8.3). If K is an empty clause, the initial formula $F_1 \wedge F_2$ is unsatisfiable. In this case, *PrvRed* returns K as a solution (line 8). Otherwise, $K \Rightarrow C_{trg}$ and K contains at least one literal of C_{trg}. Besides, $(F_1 \cup F_2) \setminus \{C_{trg}\} \dot{\Rightarrow} K$. *START* removes C_{trg} from F_1 and begins a new iteration (line 9).

8.2 High-level view of *PrvRed*

The algorithm of *PrvRed* is similar to that of a SAT-solver [32]. *PrvRed* makes decision assignments and runs Boolean Constraint Propagation (BCP). In particular, *PrvRed* uses the notion of a **decision level** that consists of a decision assignment and the implied assignments derived by BCP. The decision level number 0 is an exception: it consists only of implied assignments. When a backtracking condition is met (e.g. C_{trg} is blocked), *PrvRed* analyzes the situation and generates a new clause. Then *PrvRed* backtracks to the lowest decision level where an implied assignment can be derived from the learned clause. (Like a SAT-solver backtracks to the lowest level where the conflict clause is unit.)

However, there are important differences between *PrvRed* and a SAT-solver. First, the goal of *PrvRed* is to prove C_{trg} redundant rather than find a satisfying assignment. So, it enjoys a richer set of backtracking conditions. This set is *complete* i.e. a backtracking condition is always met when assigning variables of $X \cup Y$. (More details are given in Subsection 9.2 and Appendix C.3.) Second, *PrvRed* learns both conflict and *non-conflict* clauses (see the next subsection). The type of derived clause depends on the backtracking condition met during BCP. Third, when C_{trg} becomes unit, *PrvRed* recursively calls itself to prove redundancy of clauses of $F_1 \wedge F_2$ resolvable with C_{trg} on the unassigned variable (see Subsection 9.3). Fourth, due to recursive calls, *PrvRed* backtracks differently from a SAT-solver.

8.3 Clauses generated by *START*

PrvRed backtracks when it is able to generate a clause K implying C_{trg} in the current subspace $\vec{a}$ i.e. $K_{\vec{a}} \Rightarrow (C_{trg})_{\vec{a}}$. We call K *a certificate clause* (or just **a**

certificate for short) because it certifies the redundancy of C_{trg} in subspace[7] $\vec{a}$. We will refer to K as **a witness-certificate** if K is derived without using C_{trg}. Then K is es-implied by $(F_1 \cup F_2) \setminus \{C_{trg}\}$ and so, adding K to $F_1 \wedge F_2$ is optional. We will call K **a participant-certificate** if it is derived using clause C_{trg}. In this case, one cannot guarantee that K is es-implied (let alone implied) by $(F_1 \cup F_2) \setminus \{C_{trg}\}$. So to claim that C_{trg} is redundant in $\exists X[F_1 \wedge F_2]$ in subspace $\vec{a}$, one *has to* add K to $F_1 \wedge F_2$.

We will refer to K as **a conflict certificate** for C_{trg} in subspace $\vec{a}$ if $K_{\vec{a}}$ is an empty clause. (In this case, $K_{\vec{a}}$ trivially implies $(C_{trg})_{\vec{a}}$.) If $K_{\vec{a}}$ implies $(C_{trg})_{\vec{a}}$ not being empty, it is called **a non-conflict certificate**. In this case, every literal $l(w)$ of K where variable w is not assigned by $\vec{a}$ is present in C_{trg}.

Definition 13. *Let certificate K state redundancy of clause C_{trg} in a subspace. We will refer to the clause consisting of the literals of K that are not in C_{trg} as the* ***conditional of K****.*

If the conditional of K is falsified in subspace $\vec{a}$, then $K_{\vec{a}} \Rightarrow (C_{trg})_{\vec{a}}$. One can derive an implied assignment from K when its *conditional* is unit like this is done by a SAT-solver when a clause becomes unit (see the example below).

Example 1. Let $K = \overline{y}_1 \vee x_2 \vee x_3$ certify redundancy of $C_{trg} = x_3 \vee x_7$ in subspace $\vec{a} = (y_1 = 1, x_2 = 0)$ i.e. $K_{\vec{a}} \Rightarrow (C_{trg})_{\vec{a}}$. The conditional of K is $\overline{y_1} \vee x_2$. Suppose $y_1 = 1$ but x_2 is unassigned yet. Then the conditional of K is unit. Since K proves C_{trg} redundant if $y_1 = 1, x_2 = 0$, one can derive the assignment $x_2 = 1$ directing search to a subspace where C_{trg} is not proved redundant yet.

A conflict certificate derived by *PrvRed* can be either a participant-certificate or a witness-certificate (depending on whether C_{trg} is involved in the conflict at hand). At the same time, all non-conflict certificates generated by *PrvRed* are witness-certificates. So, *PrvRed* does not have to store non-conflict certificates by adding them to $F_1 \wedge F_2$. On the other hand, storing and reusing non-conflict certificates in different branches of the search tree can dramatically boost the performance of a PQE solver. Nevertheless, for the sake of simplicity, in the current implementation of *START*, the full power of non-conflict certificates is not exploited yet. The *PrvRed* procedure does not add non-conflict certificates to the formula and so, these certificates are not reused at all for more detail). (The *Recurse* procedure described in Subsection 9.3 does add the non-conflict certificates returned by *PrvRed* but they are removed from the formula as soon as *Recurse* terminates.)

8.4 Generation of a clause implying C_{trg} when C_{trg} is blocked

Let C_{trg} be blocked in a subspace. Then *PrvRed* generates a clause K implying C_{trg} in this subspace such that $(F_1 \cup F_2) \setminus \{C_{trg}\} \overset{\exists}{\Rightarrow} K$ with respect to Y. (The

[7] If $\vec{a}$ satisfies C_{trg}, the latter is redundant in subspace $\vec{a}$. However,*PrvRed* never proves redundancy of C_{trg} by making an assignment satisfying its literal. So, we assume here that $\vec{a}$ does not satisfy C_{trg}.

clause K is an example of a non-conflict witness-certificate.) This is the only case where *START* builds a clause that, in general, is *not* implied by the current formula $F_1 \wedge F_2$. Below, we denote $F_1 \wedge F_2$ as F.

Proposition 2. *Let $F(X,Y)$ be a formula and C_{trg} be a clause of F. Let C_{trg} be blocked in F at $w \in X$ with respect to Y in subspace $\vec{q}$ where $w \notin Vars(\vec{q})$. Let $l(w)$ be the literal of w that is in C_{trg}. Let C_{trg} be resolvable with all clauses of F with $\overline{l(w)}$. Let $K = C^* \vee l(w)$ where C^* is the longest clause falsified by $\vec{q}$. Then $K \Rightarrow C_{trg}$ in subspace $\vec{q}$ and $F \setminus \{C_{trg}\} \ddot{\Rightarrow} K$ with respect to Y.*

The clause C_5 of Section 6 is an example of a clause built using Proposition 2. One can show that the clause K of Proposition 2 is es-implied not only by $F \setminus \{C_{trg}\}$ but by F as well (see Appendix B.2). If there is a non-empty set $R \subseteq F$ of clauses with literal $\overline{l(w)}$ that are *unresolvable* with C_{trg} on w, one needs to add to K extra literals of C_{trg}. Namely, one needs to add a subset of literals of C_{trg} guaranteeing that K is *also* unresolvable on w with every clause of R unsatisfied by $\vec{q}$ (see Appendix B.1).

9 *START* In More Detail

9.1 Description of the *PrvRed* procedure

The pseudo-code of *PrvRed* is shown in Fig 2. The objective of *PrvRed* is to prove the current target clause C_{trg} redundant in the subspace specified by an assignment $\vec{a}$ to $X \cup Y$. First, *PrvRed* stores the initial value of $\vec{a}$ that is used later to identify the termination time of *PrvRed* (line 1).

The main work is done in a while loop that is similar to the main loop of a SAT-solver [32]. The operation of *PrvRed* in this loop is partitioned into two parts separated by the dotted line. The first part (lines 3-7) starts with picking a new decision assignment $v{=}b$ and adding it to the assignment queue Q (lines 3-4). Here $v{\in}(X{\cup}Y)$ and $b \in \{0,1\}$. The variables of Y are the first to be assigned[8] by *PrvRed*. So $v \in X$, only if all variables of Y are assigned. If $v \in Vars(C_{trg})$, then $v = b$ is picked so as to *falsify* the corresponding literal of C_{trg}. (C_{trg} is obviously redundant in a subspace where it is satisfied.) Then *PrvRed* calls the *BCP* procedure. If no backtracking condition is met in *BCP*, *PrvRed* begins a new iteration (lines 6-7). Otherwise, *PrvRed* goes to the second part of the while loop where it backtracks (since C_{trg} is proved redundant in subspace $\vec{a}$). In particular, *PrvRed* backtracks if *BCP* returns a clause C_{imp} implying C_{trg} in subspace $\vec{a}$ or a clause C_{fls} falsified by $\vec{a}$.

[8] The final goal of PQE is to derive clauses depending only on Y that make the X-clauses of F_1 redundant. *START* facilitates achieving this goal by assigning variables of X after those of Y. So, when backtracking, the variables of X are resolved out *before* those of Y.

```
// ξ denotes (F1, F2, X, C_trg)
// φ denotes (ans, F1, F2, a⃗)
//
PrvRed(F1,F2 ‖ C_trg,X,a⃗){
1  a⃗_init := a⃗
2  while (true) {
3     (v, b) := MakeDec(ξ)
4     UpdQueue(Q ‖ v, b) }
5     (ans, C_imp, C_fls) := BCP(Q,a⃗‖ ξ)
6     if (ans = NoBcktr)
7        continue
   - - - - -
8     (K, K_aux) := Lrn(φ, C_imp, C_fls)
9     if (Confl(K) ∨ (K_aux ≠ nil))
10       Store(F1, F2 ‖ K, K_aux)
11    Backtrack(a⃗ ‖ K)
12    if (a⃗ ⊆ a⃗_init) return(K)
13    UpdQueue(Q ‖ a⃗, K)}}
```

Fig. 2: The *PrvRed* procedure

PrvRed starts the second part (lines 8-13) with calling the *Lrn* procedure to generate a certificate K (line 8). In the situation described in Subsection 10.5, *Lrn* also derives an auxiliary certificate K_{aux}. If K is a conflict certificate[9], it is added to the formula (lines 9-10). If *Lrn* returns K_{aux}, it is also added to the formula because K_{aux} is always a conflict certificate. If a clause of F_1 is used in generation of K, the latter is added to F_1. Otherwise, K is added to F_2. The same rule applies to storing K_{aux}.

After that, *PrvRed* backtracks (line 11). If *PrvRed* reaches the subspace $\vec{a}_{init}$, the redundancy of C_{trg} in the required subspace is proved and *PrvRed* terminates (lines 12). Otherwise, an assignment is derived from K and added to the queue Q (line 13). This derivation is due to the fact that after backtracking, the conditional of K or clause K itself become *unit* (see Subsection 10.3).

9.2 BCP

The main loop of *BCP* consists of two parts separated by the dotted line in Fig. 3. *BCP* starts the first part (lines 2-7) with extracting an assignment $w = b$ from the assignment queue Q (line 2). It can be a decision assignment or one derived[10] from a clause C of the original formula $F_1 \wedge F_2$ or a certificate K. If $w = b$ is derived from the target clause C_{trg}, *BCP* calls *Recurse* to prove redundancy of clauses that can be resolved with C_{trg} on w (line 4). Why and how this is done is explained in Subsection 9.3. *BCP* processes the unit clause C_{trg} only if *all other* assignments of the queue Q have been already made. Calling *Recurse* modifies $F_1 \wedge F_2$ so that C_{trg} gets blocked in subspace $\vec{a}$ or a clause falsified by $\vec{a}$ is added. (In either case, C_{trg} is proved redundant in subspace $\vec{a}$.) If $w = b$ is *not* derived from C_{trg}, *BCP* just makes this assignment and updates the queue Q by checking if new unit clauses have appeared (lines 6-7).

In the second part (lines 8-16), *BCP* checks the backtracking conditions. First, *BCP* checks if the queue Q contains an assignment derived from a clause C_{imp} that satisfies C_{trg} (line 8). The latter means that C_{imp} implies C_{trg} in subspace $\vec{a}$. If so, *BCP* terminates returning the backtracking condition *ImplTrg*

[9] As mentioned earlier, in the current version of *PrvRed*, *non-conflict* certificates are not added to formula $F_1 \wedge F_2$.

[10] For implied assignments, *START* maintains the same order constraint as for decision ones. Namely, an implied assignment to a variable of X is made only if all variables of Y are already assigned.

(lines 9-10). Otherwise, *BCP* checks if a clause C_{fls} of $F_1 \wedge F_2$ is falsified (lines 11-13). If C_{fls} exists, one can make C_{trg} redundant in the current subspace by adding a conflict clause. Otherwise, *BCP* checks if C_{trg} is blocked (lines 14-16). If so, then an es-clause C_{imp} implying C_{trg} in subspace $\vec{a}$ is generated (see Subsection 8.4). If *BCP* empties the queue Q without meeting a backtracking condition, it terminates returning *NoBacktr*.

9.3 The case where C_{trg} becomes unit

```
// η denotes (C_trg,X,a⃗,w)
//
BCP(Q, a⃗‖F_1, F_2, X, C_trg) {
1  while (Q ≠ ∅) {
2     (w, b, C, K) := Pop(Q ‖ )
3     if (C = C_trg)
4        return(Recurse(F_1,F_2‖η))
5     else {
6        Assign(a⃗, Q ‖w,b,C,K)
7        UpdQueue(Q‖F_1,F_2,a⃗)}
   - - - - -
8     C_imp := Implied(Q, C_trg)
9     if (C_imp ≠ nil)
10       return(ImplTrg, C_imp, nil)
11    C_fls := CheckCnfl(F_1,F_2, a⃗)
12    if (C_fls ≠ nil)
13       return(Cnfl, nil, C_fls)
14    C_imp := Blk(F_1,F_2,a⃗,C_trg)
15    if (C_imp ≠ nil)
16       return(BlkTrg, C_imp, nil)}
17 return(NoBacktr, nil, nil)}
```

Fig. 3: The *BCP* procedure

Now, we describe what *PrvRed* does when C_{trg} becomes unit. Consider the following example. Let $C_{trg} = y_1 \vee x_2$ and $y_1 = 0$ in the current assignment $\vec{a}$ and x_2 not be assigned yet. Since $\vec{a} \cup \{x_2 = 0\}$ falsifies C_{trg}, a SAT-solver would derive $x_2 = 1$. However, the goal of *PrvRed* is to prove C_{trg} *redundant* rather than check if $F_1 \wedge F_2$ is satisfiable. The fact that C_{trg} is falsified in subspace $\vec{a} \cup \{x_2 = 0\}$ says nothing about whether it is redundant there.

To address the problem above, *START* recursively calls *PrvRed* to prove that every clause of $F_1 \wedge F_2$ that contains the literal $\overline{x}_2$ and is *resolvable* with C_{trg} on x_2 is redundant in subspace $\vec{a} \cup \{x_2 = 1\}$. This results in proving redundancy of C_{trg} in one of two ways. First, a clause falsified in subspace $\vec{a}$ is derived. Adding it to $F_1 \wedge F_2$ makes C_{trg} redundant in this subspace. Second, *PrvRed* proves every clause resolvable with C_{trg} on x_2 redundant in subspace $\vec{a} \cup \{x_2 = 1\}$ without generating a clause falsified by $\vec{a}$. Then C_{trg} is *blocked* at variable x_2 in subspace $\vec{a}$ (see Definition 12) and thus is redundant there.

The recursive calls of *PrvRed* are made by procedure *Recurse* (line 4 of Fig. 3). *Recurse* starts with setting to $\emptyset$ the set G meant to accumulate non-conflict certificates. The main body of *Recurse* consists of a while loop (lines 2-13 of Fig. 4). First, *Recurse* selects a clause B resolvable with C_{trg} on variable w that is neither satisfied nor proved redundant yet (line 3). Here w is the only unassigned variable of C_{trg}. If B does not exist, *Recurse* breaks the loop (line 4). Otherwise, it calls *PrvRed* to check the redundancy of B in subspace $\vec{a^*}$. The assignment $\vec{a^*}$ is obtained from $\vec{a}$ by adding $w = d$ satisfying C_{trg} where $d \in \{0, 1\}$. *PrvRed* generates a certificate K stating the redundancy of B in subspace $\vec{a^*}$ (line 6). If K contains the variable w, the former is replaced with the resolvent of K and C_{trg} on w (line 8). So, the new clause K does not depend on w and K will not be selected as a new clause to be proved redundant in line

3. If K is a conflict certificate, after removing variable w it is falsified by $\vec{a}$. So *Recurse* removes[11] the certificates accumulated in G and terminates (lines 9-11).

If K is a non-conflict certificate, *Recurse* adds it to G and $F_1 \wedge F_2$ and starts a new iteration (lines 12-13). Adding K implying B to the formula means the clause B can be ignored until *Recurse* is done looping. Once looping is over, every clause B resolvable with C_{trg} on w is either satisfied or proved redundant in subspace $\vec{a^*}$. Hence, C_{trg} is blocked at w. *Recurse* generates a clause C_{imp} implying C_{trg} in subspace $\vec{a}$, removes the certificates of G and terminates (lines 14-16). The clause C_{imp} is built using Proposition 2 and the certificates of G (see the example below).

```
// F denotes F1 ∪ F2
//
Recurse(F ‖ C_trg,X,a⃗,w){
1  G := ∅
2  while (true) {
3     B := SelCls(F,C_trg,w)
4     if (B = nil) break
5     a⃗* := a⃗ ∪ {w = d}
6     K := PrvRed(F‖ B,X,a⃗*)
7     if (w ∈ Vars(K))
8        Resolve(K ‖ C_trg,w)
9     if (Confl(K)) {
10       F := F \ G
11       return(Cnfl, nil, K)}
12    G := G ∪ {K}
13    F := F ∪ {K} }
14 C_imp := Blk(F, G,a⃗, C_trg, w)
15 F := F \ G
16 return(Blk, C_imp, nil)}
```

Fig. 4: The *Recurse* procedure

Example 2. Let $X = \{x_1, x_2, x_3, \dots\}$ and $Y = \{y_1, y_2, y_3\}$. Suppose $F_1 \wedge F_2$ contains (among others) the clauses $C_1 = y_3 \vee x_1$, $C_2 = \overline{x}_1 \vee x_2$, $C_3 = \overline{x}_1 \vee x_3$. Let C_1, C_2, C_3 be the only clauses of $F_1 \wedge F_2$ depending on x_1. Let C_1 be the current target clause and the current assignment $\vec{a}$ to $X \cup Y$ be equal to $(y_1 = 0, y_2 = 0, y_3 = 0)$. Then the target clause C_1 turns into the unit clause x_1 in subspace $\vec{a}$. So, *BCP* calls the *Recurse* procedure to prove redundancy of C_2 and C_3 in subspace $\vec{a^*} = \vec{a} \cup \{x_1 = 1\}$. That is the only free variable of C_1 in subspace $\vec{a}$ is assigned the value satisfying C_1.

Suppose that the call of *PrvRed* made by *Recurse* to prove C_2 redundant in subspace $\vec{a^*}$ returns the non-conflict certificate $K' = y_1 \vee x_2$ (implying C_2 in subspace $\vec{a^*}$). Suppose that another call of *PrvRed* proves redundancy of C_3 in subspace $\vec{a^*}$ by deriving the non-conflict certificate $K'' = y_2 \vee x_3$ (implying C_3 in subspace $\vec{a^*}$). Since both C_2 and C_3 are redundant in subspace $\vec{a^*}$, the target clause C_1 is blocked in subspace $\vec{a}$ at variable x_1. In reality, C_1 is blocked even in subspace $(y_1 = 0, y_2 = 0)$ because K' and K'' still imply C_2 and C_3 in this subspace. Then one can use Proposition 2 to derive the clause $K = y_1 \vee y_2 \vee x_1$. It consists of the literals falsified by $(y_1 = 0, y_2 = 0)$ and the literal x_1 of the target clause C_1. On one hand, $(F_1 \cup F_2) \setminus \{C_{trg}\} \ddot{\Rightarrow} K$. On the other hand, $K_{\vec{a}} \Rightarrow (C_1)_{\vec{a}}$. So K certifies the redundancy of C_1 in subspace $\vec{a}$.

[11] Note that *Recurse* does not remove the clauses of the original formula even if they are proved redundant. So any non-conflict certificate added to the current formula by *Recurse* can be safely removed (since such a removal preserves equisatisfiability).

10 Generation Of New Clauses And Backtracking

10.1 Generation of new clauses

When *BCP* reports a backtracking condition, the *Lrn* procedure generates a certificate K (line 8 of Fig 2). We will refer to the decision level where a backtracking condition is met as the **event level**. *Lrn* generates a *conflict* certificate K if a) *BCP* finds a falsified clause C_{fls} and b) no implied assignment of the event level relevant to falsifying C_{fls} is derived from a non-conflict certificate. (If this is not the case, *Lrn* may generate an auxiliary certificate K_{aux}, see Subsection 10.5.) A conflict certificate K is a participant-certificate if its derivation involves C_{trg}. (So, to guarantee that C_{trg} is redundant in the current subspace one *has to* add K to the formula.) Otherwise, K is a witness-certificate showing that C_{trg} is already redundant in the current subspace. In all other cases i.e. when C_{trg} is blocked or implied by an existing clause of the formula, *Lrn* generates a *non-conflict* certificate K. All non-conflict certificates are witness-certificates.

A conflict certificate K is built by *Lrn* as a conflict clause is constructed by a regular SAT-solver [32]. Originally, K equals the clause C_{fls} returned by *BCP* that is falsified in subspace $\vec{a}$. Then *Lrn* resolves out literals of K falsified by assignments *derived* at the event level by BCP. This procedure stops when only one literal of K is assigned at the event level. (So, after backtracking, K becomes unit and an assignment can be derived from it.).

A non-conflict certificate K is built as follows. Originally, K equals the clause C_{imp} returned by *BCP* that implies C_{trg} in subspace $\vec{a}$. *Lrn* also resolves out literals of K falsified by assignments derived at the event level. The difference here is twofold. First, in a non-conflict certificate, only the literals of its *conditional* are certainly falsified by $\vec{a}$ (see Definition 13) as opposed to a conflict certificate. Second, in a non-conflict certificate, the requirement is that only one literal of its *conditional* is assigned at the event level. (This guarantees that the conditional of K becomes unit after backtracking and an implied assignment can be derived from K.) Besides, this literal is required to be assigned by the *decision* assignment of the event level (unless it is the decision level number 0).

Example 3. Suppose $X = \{x_1, x_2, x_3, \dots\}, Y = \{y\}$ and $F_1 \wedge F_2$ contains (among others) the clauses $C_1 = y \vee x_1$, $C_2 = \overline{x}_1 \vee x_2$, $C_3 = x_2 \vee x_3$. Suppose C_3 is the current target clause and *PrvRed* makes the decision assignment $y = 0$. Since C_1 becomes unit, *BCP* derives $x_1 = 1$. Then C_2 turns into the unit clause x_2 that implies C_3. So *BCP* terminates returning the backtracking condition *ImplTrg* and clause C_2 as implying the current target C_3 (line 10 of Fig. 3). The assignment $\vec{a}$ at this point is equal to $(y = 0, x_1 = 1)$. Then the *Lrn* procedure is called to generate a non-conflict certificate K. Originally, $K = C_2$. It contains the literal $\overline{x}_1$ falsified by the assignment $x_1 = 1$ *derived* at the event level. To get rid of $\overline{x}_1$, *Lrn* resolves K and the clause C_1 from which $x_1 = 1$ is derived. The new clause K is equal to $y \vee x_2$. It implies C_3 under assignment $\vec{a}$ and only the decision assignment $y = 0$ falsifies a literal of C_3 at the event level. So K is the required certificate. Note that only the conditional of K (consisting of literal y) is falsified by $\vec{a}$. The clause K itself is not falsified by $\vec{a}$ since x_2 is unassigned.

10.2 New proof obligations

Adding an X-clause to the formula may create a new proof obligation. It is fulfilled in one of two ways. Let an X-clause of F_1 be used to build a certificate K (by resolution). Then K is added to F_1 and proved redundant by a future call of *PrvRed* in the main loop of *START* (Fig. 1, line 7). Let C be a descendant of a clause B selected by procedure *Recurse* as the new target from clauses sharing the same literal $l(w)$ of variable w (Fig. 4, line 3). That is C is one of the certificate clauses added when proving B redundant (Fig. 4, line 6). If C depends on w, it contains the literal $l(w)$. So the redundancy of C is proved by a future call of *PrvRed* (Fig. 4, line 6).

10.3 Backtracking

After generating a certificate K, *PrvRed* calls the *Backtrack* procedure (line 11 of Fig. 2). Since *Recurse* may recursively call *PrvRed* with a new target clause, *PrvRed* backtracks differently from a SAT-solver. Let d_1 denote the decision level where the current call of *PrvRed* was made. For the target clause of *PrvRed* called in the main loop of *START*, $d_1 = 0$. Let d_2 denote the largest decision level where K is unit (if K is a conflict certificate) or the conditional of K is unit (if K is a non-conflict certificate). If $d_1 < d_2$, then redundancy of C_{trg} in the required subspace is not proved yet. So *Backtrack* returns to the level d_2. This mimics what a SAT-solver does after a conflict clause is derived. However, if $d_1 >= d_2$, proving C_{trg} redundant in the required subspace is finished. If the current call of *PrvRed* is made by the *Recurse* procedure, C_{trg} loses its status of the target clause. *PrvRed* returns to level d_1 and the current call of *PrvRed* terminates. As we mentioned earlier, if K is a non-conflict certificate, it is not reused in the current version of *START*. K is kept as long as its conditional remains unit and so an assignment is derived from K. As soon as backtracking unassigns at least two literals of the conditional of K, the latter is discarded.

10.4 Conflicts involving non-conflict certificates ($C_{fls} \neq C_{trg}$)

Let C_{fls} be the clause returned by *BCP* as falsified by the current assignment $\vec{a}$. Suppose some assignments contributing to falsifying C_{fls} were derived at the event level from non-conflict certificates. Let $w = b$ be the last of such assignments. If C_{fls} is not the target clause C_{trg}, *Lrn* derives a certificate K like it does in Subsection 10.1 for a non-conflict certificate. (Only *Lrn* starts with C_{fls} instead of C_{imp}.) Since C_{trg} is not involved in generation of K, the latter is a *witness-certificate*[12].

Example 4. Let $F_1 = C_1$ where $C_1 = x_1 \vee x_2$. Let $F_2 = C_2 \wedge C_3 \wedge C_4 \wedge \ldots$ where $C_2 = y \vee \overline{x}_3$, $C_3 = y \vee \overline{x}_4$, $C_4 = x_3 \vee x_4$. Let $X = \{x_1, x_2, x_3, x_4 \ldots\}$. Let C_1

[12] Recall that if C_{trg} is unit, *PrvRed* is recursively called with a *new* target. So, no assignment is derived from C_{trg} in the *current* version of *PrvRed*. Hence, if $C_{fls} \neq C_{trg}$, then C_{trg} is *not* used in the conflict and so is redundant in subspace $\vec{a}$.

be our target clause. Suppose that *START* derived the non-conflict certificate $K^* = \overline{y} \vee x_1$ to show that C_1 is redundant in subspace $y = 1$. The conditional of K^* is the unit clause $\overline{y}$. After deriving K^*, *START* backtracks to the decision level number 0, the lowest level where the conditional of K^* is unit. Then *START* makes the implied assignment $y = 0$ turning clauses C_2, C_3 into unit. This entails derivation of assignments $x_3 = 0$ and $x_4 = 0$ falsifying the *non-target* clause C_4. So, the current assignment $\vec{a}$ equals $(y = 0, x_3 = 0, x_4 = 0)$.

In this case, *Lrn* builds a certificate K similar to generation of a non-conflict certificate (see Subsection 10.1). Only instead of clause C_{imp} one starts with C_{fls} (i.e. with a falsified clause). So, originally, K is equal to the falsified clause C_4. By resolving K with clauses C_2, C_3 and clause K^* (from which $y = 0$ was derived) one generates the certificate $K = x_1$. Since, the target clause was not used in derivation of K, the latter is a witness-certificate. Besides, $\vec{a}$ does not falsify K (because the non-conflict certificate K^* was used to produce K). So K is a non-conflict certificate.

10.5 Conflicts involving non-conflict certificates ($C_{fls} = C_{trg}$)

Now, let us consider the case $C_{fls} = C_{trg}$. Suppose K is built as in the previous subsection (where $C_{fls} \neq C_{trg}$). Since C_{trg} is involved, K is a *participant-certificate* that needs to be added to the formula. On the other hand, since non-conflict certificates are used to generate K, the latter contains some literals of C_{trg}. If K is an X-clause, adding K to the formula creates a new proof obligation. In particular, if K consists of a subset of literals of C_{trg}, one replaces the original proof obligation of showing redundancy of C_{trg} with a bigger one. (Proving redundancy of a shorter clause is harder.)

Lrn addresses the problem above by generating *two* certificates: a participant-certificate K_{aux} and a witness-certificate K. The subscript "aux" stands for "auxiliary". Adding clause K_{aux} makes C_{trg} redundant in subspace $\vec{a}$ and K is just a witness of this redundancy. K_{aux} is generated as a regular conflict clause until the assignment $w = b$ above is reached. Then *Lrn* uses K_{aux} as a starting clause falsified by $\vec{a}$ and generates a certificate K as in the case $C_{fls} \neq C_{trg}$ considered in the previous subsection.

Example 5. Let $F_1 = C_1$ where $C_1 = x_1 \vee x_2$. Let $F_2 = C_2 \wedge C_3 \wedge \dots$ where $C_2 = y \vee \overline{x}_1$, $C_3 = y \vee \overline{x}_2$. Let $X = \{x_1, x_2, \dots\}$. Let C_1 be our target clause. Suppose that *START* derived the non-conflict certificate $K^* = \overline{y} \vee x_1$ to show that C_1 is redundant in subspace $y = 1$. The conditional of K^* is the unit clause $\overline{y}$. After deriving K^*, *START* backtracks to the decision level number 0, the lowest level where the conditional of K^* is unit. Then *START* makes the implied assignment $y = 0$ turning clauses C_2, C_3 into unit. This entails derivation of assignments $x_1 = 0$ and $x_2 = 0$ falsifying the *target* clause C_1. So, the current assignment $\vec{a}$ equals $(y = 0, x_1 = 0, x_2 = 0)$.

Assume *Lrn* builds a conflict certificate K as described in Example 4. Originally, K is equal to the falsified clause C_1. By resolving K with clauses C_2, C_3 and clause K^* (from which $y = 0$ was derived) one generates the certificate

$K = x_1$. Since the target clause was used to build K the latter has to be added to the formula (i.e. K is a participant-certificate). Besides, since $C_1 \in F_1$, the clause K is added to F_1. So one replaces proving the redundancy of $C_1 = x_1 \wedge x_2$ with a harder problem of proving redundancy of $K = x_1$.

One can fix the problem above by deriving *two certificates*: the participant-certificate $K_{aux} = y$ and the witness-certificate $K = x_1$. The certificate K_{aux} is derived by resolving the falsified clause C_1 with C_2 and C_3. One can view K_{aux} as obtained by *breaking* the process of certificate generation. This break occurs when the assignment $y = 0$ (derived from the non-conflict certificate K^*) is reached. Since, the target clause C_1 is used to derive K_{aux}, the latter is a participant-certificate. Adding K_{aux} to $F_1 \wedge F_2$ makes C_1 redundant in subspace $y = 0$.

Building the certificate K can be viewed as *completing* the process of certificate generation interrupted by assignment $y = 0$. Only now the certificate K_{aux} is used as the initial clause falsified by $\vec{a}$ (instead of the original falsified clause C_1). The clause K is obtained by resolving K_{aux} (that made C_1 redundant in subspace $y = 0$) with the certificate K^* (showing redundancy of C_1 in subspace $y = 1$). Importantly, the target clause C_1 is not used when generating K. So the latter is a witness-certificate of the *global* redundancy of C_1 in $\exists X[F_1 \wedge F_2 \wedge K_{aux}]$.

11 Experimental Results

In this section, we describe some experiments with an implementation of $START$. This implementation still has two major weaknesses. First, non-conflict certificates are not reused. So $START$ does not do any learning in the subspaces where the formula is satisfiable. Second, the restriction that unquantified variables are assigned before quantified ones harms the performance of $START$. (This restriction can be relaxed, however, a discussion of such relaxation is beyond the scope of this paper.)

Since PQE is a new problem, no established set of benchmarks exists. To evaluate our implementation[13], we used the problem of *computing range reduction* [23]. We picked this problem for three reasons. First, it has an important application in the context of testing. Second, one can use this problem to show that PQE can be much easier than QE. Third, since it is a new problem, even the current implementation of $START$ can make a mark. Once $START$ matures implementation-wise, it can be applied to other problems e.g. those listed in Sections 2-3.

Let $N(X, Y, Z)$ be a combinational circuit where X, Y, Z are sets of input, internal and output variables. Suppose one applies random tests (i.e. full assignments to X) to check if N produces erroneous outputs. In many cases,

[13] As we mentioned earlier, a proof of correctness of $START$ is given in Appendix C. To get extra guarantees that our *implementation* of $START$ is correct as well, we verified its results on a large number of formulas (see Appendix E).

tests producing erroneous outputs are rare and so are hard to generate randomly. One can mitigate this problem as follows. Let formula $F(X, Y, Z)$ specify N (see Footnote 6). The **range** of N, i.e. the set of outputs produced by N, is specified by $\exists W[F]$ where $W = X \cup Y$. Let $G(X)$ be a formula such that $\exists W[G \wedge F] \equiv \exists W[F]$. Then the set of tests satisfying G **preserves the range** of N. That is for every test $\vec{x'}$ falsifying G there is a test $\vec{x''}$ *satisfying* G that produces the same output as $\vec{x'}$. The benefit of using a range preserving constraint G is that one can explore only tests satisfying G and still guarantee that every erroneous output can be produced. Checking if G preserves the range of N reduces to verifying the redundancy of G in $\exists W[G \wedge F]$ i.e. to PQE.

11.1 Description of experiments

As examples of realistic combinational circuits, we used 555 transition relations of sequential circuits from the HWMCC-10 set[14]. To decrease the size of the problem to consider, we used the following idea. Let N be a subcircuit of M such that a) N has the same input variables as M and b) the output gates of N form a *cut* in M. Then if an input constraint G preserves the range of N, it also preserves the range of M. (However, the opposite is not necessarily true.) So, in the experiments, we replaced the circuit M specifying a transition relation with a subcircuit N described above. More details are given in Appendix D.

Let formula $F(X, Y, Z)$ specify a circuit N. The input constraints we tried in experiments were generated as follows. First, we selected a subset X' of input variables X and for every variable $x \in X'$ we considered the two input constraints specified by unit clauses x and $\overline{x}$. (The construction of X' is described in Appendix D. The size of X' was limited by 50. We used X' instead of X just to make experiments less time consuming.) So, for every circuit N, we generated $2 * |X'|$ problems of checking if $l(x)$ was redundant in $\exists W[l(x) \wedge F]$ where $W = X \cup Y$ and $l(x)$ is x or $\overline{x}$. The redundancy of $l(x)$ means that fixing x at the value satisfying $l(x)$ preserves the range of N. So it is sufficient to generate only tests satisfying $l(x)$. Checking redundancy of $l(x)$ is performed in two steps. First, *START* generates a formula $H(Z)$ such that $H \wedge \exists W[F] \equiv \exists W[l(x) \wedge F]$. Then one searches for a clause $C \in H$ that is *not* implied by F (see Remark 1). If C does not exist, then $l(x)$ is redundant and the range of N remains intact if x is fixed at the value satisfying $l(x)$. Otherwise, the range changes.

The results of *START* are given in Table 1. The first column shows the number of circuits. The second column gives the total number of PQE problems i.e. the sum of $2 * |X'|$ over 555 circuits. The third column shows the number of PQE problems solved by *START* in the time limit of 1 second. The fourth column gives the number of solved problems where $l(x)$ was redundant in $\exists W[l(x) \wedge F]$ (and so one could fix the value of x without changing the range of N). The last

[14] The HWMCC-10 set consists of 758 benchmarks encoding safety properties. Some benchmarks specify different properties of the same sequential circuit. So the number of different transition relations (555) is smaller than that of benchmarks.

column shows the number of circuits N where the value of at least one variable $x \in X'$ can be fixed without changing the range of N.

11.2 Comparing complete and partial QE

Table 1: Results of *START*. The time limit is 1 sec.

#circs	#probs	#slvd	#succ. constr.	
			#probs	#circs
555	36,556	19,162	1,902	105

As we mentioned above, the range of a circuit N specified by formula $F(X, Y, Z)$ is defined as $\exists W[F]$ where $W = X \cup Y$. This range can be computed by finding formula $F^*(Z)$ such that $F^* \equiv \exists W[F]$ i.e. by QE. To check if fixing a variable $x \in X'$ at the value satisfying $l(x)$ preserves the range of N, one can perform the following two steps. First, the ranges of unconstrained and constrained N are computed by solving QE for $\exists W[F]$ and $\exists W[l(x) \wedge F]$ respectively. Second, the results of QE are checked for equivalence. On the other hand, one can solve *the same problem* (of testing if the constraint $l(x)$ affects the range of N) by PQE i.e. by checking if $l(x)$ is redundant in $\exists W[l(x) \wedge F]$ as described in Subsection 11.1.

In this subsection, we compare solving the problem at hand by QE and PQE. We consider three QE tools. The first tool is *START* itself. (One can always use a PQE algorithm to perform complete QE by taking *all* clauses with quantified variables out of the scope of quantifiers.) The second tool uses BDDs and is based on the package CUDD [38]. The third tool called CADET [35] is a highly efficient SAT-based algorithm for computing Skolem functions.

Table 2: Comparison of QE and PQE in the number of *unsolved* circuits (out of 555)

time	QE			PQE
limit (s.)	start	bdds	cadet	start
1	503	495	410	**83**
10	465	451	361	**21**
60	431	429	298	**14**

QE and PQE are compared in Table 2. The first column gives the time limit. The following four columns show the number of *unsolved* circuits (out of 555) for QE (*START*, BDDs, CADET) and PQE (*START*). We considered a circuit as unsolved if none of the $2 * |X'|$ problems were finished in the time limit. Let T_{con} (respectively T_{unc}) denote the runtime for computing the range of constrained (respectively unconstrained) circuit by QE. We used $T_{con} + T_{unc}/(2 * |X'|)$ as the runtime for solving one problem by QE (because the same range of the *unconstrained* circuit was used for all $2 * |X'|$ constrained circuits and so it was computed *only once*). We ignored the runtime required to check the equivalence of ranges of constrained and unconstrained circuits. Note that $T_{con} + T_{unc}/(2 * |X'|) \approx T_{con}$. So we essentially compare the run time of QE and PQE on formula $\exists W[l(x) \wedge F]$ where QE takes the entire formula out of the scope of quantifiers and PQE takes out only $l(x)$.

Table 2 shows that CADET and BDDs outperform *START* used as a QE algorithm. This can be attributed to two factors. First, CADET and BDDs represent the resulting formula *implicitly* via introducing new variables whereas

START generates its result in terms of the output variables of N. Second, as we mentioned earlier, the current implementation of *START* has two major weaknesses. However, even the best QE tool is drastically outperformed by *START* when it solves the problem at hand by PQE. In particular, PQE has fewer unsolved circuits with the time limit of 1 second (83 circuits) than the QE tools with the time limit of 60 seconds (431, 429 and 298 circuits for *START*, BDDs and CADET respectively).

Importantly, there is no straightforward way to use CADET and BDDs for PQE. The reason why *START* can easily perform both QE and PQE is that it employs redundancy based reasoning. This type of reasoning can be called *structural* because redundancy is a structural rather than a semantic property. (Redundancy of a clause in a formula cannot be expressed in terms of the truth table of this formula. In particular, redundancy of a clause C in formula G' does not entail redundancy of C in formula G'' logically equivalent to G'.) On the other hand, CADET and BDDs employ *semantic* reasoning. A BDD represents the truth table of a formula as a network of multiplexers whereas CADET does the same via a set of Skolem functions.

11.3 Comparing *START* and *DS-PQE*

DS-PQE [24] is the only PQE algorithm we are aware of (other than *START*). It is based on the machinery of D-sequents [22,20], where a D-sequent is a record claiming redundancy of a clause in a specified subspace. The main difference between *DS-PQE* and *START* is that the former proves redundancy of *many* target clauses at once. (So, *DS-PQE* backtracks only if *all* target clauses are proved redundant in the current subspace, which may lead to generating deep search trees.) *DS-PQE* and *START* are similar in that they do not reuse D-sequents and non-conflict certificates respectively. However, as we mentioned in the introduction, reusing D-sequents is very expensive, i.e. it is problematic *in principle*. On the other hand, reusing certificates of all kinds is cheap.

Table 3: *START* versus *DS-PQE*. The time limit is 1 sec.

#circs	#same	#less	#more
555	200	54	301

In Table 3 we compare *START* and *DS-PQE*. The first column gives the number of circuits used in this experiment. The second column shows the number of circuits where *START* and *DS-PQE* solved the same number of problems (out of $2*|X'|$) in the time limit of 1 second per problem. The last two columns give the number of circuits where *START* solved less and more PQE problems per circuit than *DS-PQE*. Table 3 demonstrates that *START* outperforms *DS-PQE*. This is very encouraging taking into account that the current implementation of *START* can be drastically improved.

12 Some Background

In this section, we discuss some research relevant to our approach to PQE. Information on *complete* QE for propositional logic can be found in [10,11] (BDD based) and [33,28,15,27,9,29,7,6,35] (SAT based).

Making clauses of a formula redundant by adding resolvents is routinely used in pre-processing [13,5] and in-processing [25] phases of QBF/SAT-solving. Identification and removal of blocked clauses is also an important part of formula simplification [26]. The difference between these approaches and our is that the former are aimed at formula *optimization* whereas the latter employs redundancy based *reasoning*.

The signal-based reasoning routinely used for *circuits* [1] can be simulated by resolution enhanced by blocked clauses [26]. PQE allows to extend signal-based reasoning to *arbitrary* formulas. Consider taking A out of the scope of quantifiers in $\exists X[A(X,Y) \wedge B(X,Y)]$. Here A can be viewed as a "signal", B as a "circuit", resolvents of A and B describe "signal propagation" and the variables of Y are the "output variables".

The most successful verification methods employ *incremental* computing. For instance, checking the equivalence of *similar* circuits can be very efficient [3,30]. In property checking, great efficiency is achieved by computing only a *subset* of all unreachable states (specified by the property at hand) [8]. Due to its inherent incrementality, PQE facilitates the design of algorithms for incremental computations. For instance, PQE can be naturally applied to equivalence checking [18]. Another example of using incrementality of PQE is property generation [14,21] (see Section 3).

As we mentioned earlier, due to completeness of resolution [2], if a clause C is redundant in formula F, one can always derive a clause K such that $K \Rightarrow C$ and $(F \setminus \{C\}) \Rightarrow K$. Our algorithm *START* shows that resolution enhanced by blocked clauses is *complete* with respect to proving redundancy of a clause C in $\exists X[F]$. If C is redundant, one can always derive a clause K such that $K \Rightarrow C$ and $(F \setminus \{C\}) \ddot{\Rightarrow} K$. (We are unaware of anybody making this claim before.)

One of the challenges of operating on quantified formulas is the necessity to deal with a large number of satisfiable formulas. Consider for instance the QE problem specified by $\exists X[F(X,Y)]$. To find a solution to this problem, one needs to check satisfiability of every formula $F_{\vec{y}}$ where $\vec{y}$ is a full assignment to Y. The problem here is that to prove $F_{\vec{y}}$ unsatisfiable it suffices to find an unsatisfiable *subset* of clauses whereas a satisfying assignment has to satisfy *every* clause of $F_{\vec{y}}$. In other words, a satisfying assignment is *global* whereas an unsatisfiable core is *local*. (For infinite formulas, this fact is not trivial and is expressed by the compactness theorem [36].) This makes enumeration of satisfiable formulas $F_{\vec{y}}$ very hard as opposed to unsatisfiable formulas that often share the same unsatisfiable core. Some solutions addressing this issue have been suggested in the past (e.g. see [37]). Redundancy based reasoning provides a radical solution to this problem by *avoiding* generation of satisfying assignments. Importantly, *START* often backtracks *long before* a satisfying assignment is reached. This is possible because, in many cases, redundancy of a clause can be proved *locally*.

13 Conclusions

We consider Partial Quantifier Elimination (PQE) on propositional CNF formulas with existential quantifiers. In PQE, only a small part of the formula is taken out of the scope of quantifiers. In many verification problems one can use PQE instead of *complete* QE. We solve PQE by redundancy based reasoning. Our main conclusions are as follows. First, by using a one-target-at-a-time approach one can solve PQE by a SAT-solver-like algorithm. Second, such an algorithm has a good chance of being quite efficient. Third, as we show experimentally by the example of computing range reduction, PQE can be much simpler than QE.

References

1. M. Abramovici, M. Breuer, and A. Friedman. *Digital Systems Testing and Testable Design*. John Wiley & Sons, 1994.
2. L. Bachmair and H. Ganzinger. Resolution theorem proving. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 2, pages 19–99. North-Holland, 2001.
3. C. Berman and L. Trevillyan. Functional comparison of logic designs for vlsi chips. In *ICCAD*, pages 456–459, 1989.
4. 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.
5. A. Biere, F. Lonsing, and M. Seidl. Blocked clause elimination for qbf. CADE-11, pages 101–115, 2011.
6. N. Bjorner and M. Janota. Playing with quantified satisfaction. In *LPAR*, 2015.
7. N. Bjorner, M. Janota, and W. Klieber. On conflicts and strategies in qbf. In *LPAR*, 2015.
8. A. R. Bradley. Sat-based model checking without unrolling. In *VMCAI*, pages 70–87, 2011.
9. J. Brauer, A. King, and J. Kriener. Existential quantification as incremental sat. CAV-11, pages 191–207, 2011.
10. R. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.
11. 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.
12. W. Craig. Three uses of the herbrand-gentzen theorem in relating model theory and proof theory. *The Journal of Symbolic Logic*, 22(3):269–285, 1957.
13. N. Eén and A. Biere. Effective preprocessing in sat through variable and clause elimination. In *SAT*, pages 61–75, 2005.
14. E.Goldberg. Generation of a complete set of properties. Technical Report arXiv:2004.05853 [cs.LO], 2020.
15. M. Ganai, A.Gupta, and P.Ashar. Efficient sat-based unbounded symbolic model checking using circuit cofactoring. ICCAD-04, pages 510–517, 2004.
16. A. V. Gelder. Propositional search with k-clause introduction can be polynomially simulated by resolution. In *(Electronic) Proc. 5th Int'l Symposium on Artificial Intelligence and Mathematics*, 1998.
17. E. Goldberg. Equivalence checking by logic relaxation. Technical Report arXiv:1511.01368 [cs.LO], 2015.

18. E. Goldberg. Equivalence checking by logic relaxation. In *FMCAD-16*, pages 49–56, 2016.
19. E. Goldberg. Property checking without invariant generation. Technical Report arXiv:1602.05829 [cs.LO], 2016.
20. E. Goldberg. Quantifier elimination with structural learning. Technical Report arXiv: 1810.00160 [cs.LO], 2018.
21. E. Goldberg. On verifying designs with incomplete specification. Technical Report arXiv:2004.09503 [cs.LO], 2020.
22. E. Goldberg and P. Manolios. Quantifier elimination via clause redundancy. In *FMCAD-13*, pages 85–92, 2013.
23. E. Goldberg and P. Manolios. Bug hunting by computing range reduction. Technical Report arXiv:1408.7039 [cs.LO], 2014.
24. E. Goldberg and P. Manolios. Partial quantifier elimination. In *Proc. of HVC-14*, pages 148–164. Springer-Verlag, 2014.
25. M. Järvisalo, M. Heule, and A. Biere. Inprocessing rules. IJCAR-12, pages 355–370, 2012.
26. Matti Järvisalo, Armin Biere, and Marijn Heule. Blocked clause elimination. In *TACAS*, pages 129–144, 2010.
27. J. Jiang. Quantifier elimination via functional composition. In *Proceedings of the 21st International Conference on Computer Aided Verification*, CAV-09, pages 383–397, 2009.
28. H. Jin and F.Somenzi. Prime clauses for fast enumeration of satisfying assignments to boolean circuits. DAC-05, pages 750–753, 2005.
29. W. Klieber, M. Janota, J.Marques-Silva, and E. Clarke. Solving qbf with free variables. In *CP*, pages 415–431, 2013.
30. A. Kuehlmann and F. Krohm. Equivalence Checking Using Cuts And Heaps. *DAC*, pages 263–268, 1997.
31. O. Kullmann. New methods for 3-sat decision and worst-case analysis. *Theor. Comput. Sci.*, 223(1-2):1–72, 1999.
32. J. Marques-Silva and K. Sakallah. Grasp – a new search algorithm for satisfiability. In *ICCAD-96*, pages 220–227, 1996.
33. K. McMillan. Applying sat methods in unbounded symbolic model checking. In *Proc. of CAV-02*, pages 250–264. Springer-Verlag, 2002.
34. K. McMillan. Interpolation and sat-based model checking. In *CAV-03*, pages 1–13. Springer, 2003.
35. M. Rabe. Incremental determinization for quantifier elimination and functional synthesis. In *CAV*, 2019.
36. R. Smullyan. *First-order logic*. Dover publications, 1993.
37. L. Zhang. Solving qbf with combined conjunctive and disjunctive normal form. In *Proc. of AAAI'06, vol. 1*, pages 143–149. AAAI Press, 2006.
38. CUDD package, `https://github.com/ivmai/cudd`.

Appendix

A Incomplete Specification And Unwanted Property

In this appendix, we give an example of an unwanted property that can be derived by PQE. (This example is borrowed from [14].) Consider the design of a

combinational circuit called a *sorter*. It accepts m r-bit numbers ranging from 0 to 2^r-1, sorts them, and outputs the result. Let X and Z be sets of input and output variables of the sorter respectively. Let $\mathbb{x}_1,\ldots,\mathbb{x}_m$ and $\mathbb{z}_1,\ldots,\mathbb{z}_m$ be numbers specified by input $\vec{x}$ and output $\vec{z}$ respectively. The properties $P'(Z)$ and $P''(X,Z)$ below form a *complete* specification of the sorter.

- $P'(\vec{z})=1$ iff $\mathbb{z}_1 \leq \cdots \leq \mathbb{z}_m$,
- $P''(\vec{x},\vec{z})=1$ iff $\mathbb{z}_1,\ldots,\mathbb{z}_m$ is a permutation of $\mathbb{x}_1,\ldots,\mathbb{x}_m$.

Suppose the designer uses an *incomplete* specification $\mathbb{P}$ consisting only of the property P'. (In this case, incompleteness of $\mathbb{P}$ is just a mistake. For more complex combinational circuits or more complex systems, e.g. sequential circuits or software, forming a complete specification may be simply *infeasible*.) Let $N(X,Y,Z)$ be an *implementation* of the sorter and $F(X,Y,Z)$ be a formula describing the functionality of N. Assume $F \Rightarrow P'$ i.e. N satisfies the specification $\mathbb{P}$. Suppose N is buggy. Namely, let $\mathbb{z}_1=0$ for every output of N. (This does not contradict $F \Rightarrow P'$, since $\mathbb{z}_i \geq 0$, $1 < i \leq m$.)

Let $C \in F$ relate to the part of N responsible for the bug above. Suppose by taking the clause C out of the scope of quantifiers in $\exists Y[F]$ (i.e. by PQE) one generates a property $Q(X,Z)$. Let Q be falsified by the outputs $\vec{z}$ where $\mathbb{z}_1=b$ and b is a constant $1 \leq b \leq 2^r-1$. (The circuit N obviously meets Q because $\mathbb{z}_1=0$ in all outputs of N.) On one hand, $P' \not\Rightarrow Q$. Indeed, P' is satisfied by an assignment $\vec{z}$ where $\mathbb{z}_1 \leq \cdots \leq \mathbb{z}_m$ and $\mathbb{z}_1=b$. So, derivation of Q proves the specification $\mathbb{P}$ *incomplete*. On the other hand, Q is an *unwanted* property of N since, in a correct sorter, $\mathbb{z}_1$ can take any value from 0 to 2^r-1. So, derivation of Q exposes a hole in $\mathbb{P}$ and proves N buggy.

B Proofs

B.1 Propositions from Sections 5 and 8

Lemma 1 below is used in the proof of Proposition 1.

Lemma 1. *Let $F(X)$ be a formula and $C(X)$ be a clause blocked in F at w. Then $F \ddot{\Rightarrow} C$.*

Proof. To show that F es-implies C, one needs to prove that $\exists X[F \wedge C] \equiv \exists X[F]$ i.e. that $F \wedge C$ and F are equisatisfiable. Assume the contrary. Since $F \wedge C \Rightarrow F$, the only possibility here is that F is satisfiable whereas $F \wedge C$ is not. Let $\vec{x}$ be a full assignment to X satisfying F. Since $F \wedge C$ is unsatisfiable, $\vec{x}$ falsifies C. Let $\vec{x^*}$ be the assignment obtained from $\vec{x}$ by flipping the value of w. Let G be the set of clauses of F resolvable with C on w. Let $w=b$ where $b \in \{0,1\}$ satisfy C. (So w is assigned the value b in $\vec{x^*}$, because $\vec{x}$ falsifies C). As we show below, $\vec{x^*}$ satisfies $F \setminus G$. So $(F \setminus G)_{w=b}$ is satisfiable. By definition of a blocked clause, $G_{w=b}$ is es-implied by $(F \setminus G)_{w=b}$. So formula $F_{w=b}$ is satisfiable as well. Since $w=b$ satisfies C, $(F \wedge C)_{w=b}$ is satisfiable. Hence formula $F \wedge C$ is satisfiable and we have a contradiction.

Let us fulfill our obligation to prove that $\vec{x^*}$ satisfies $F \setminus G$. Assume the contrary i.e. $\vec{x^*}$ falsifies a clause D of $F \setminus G$. Assume that D does not contain the variable w. Then D is falsified by the assignment $\vec{x}$ and hence the latter does not satisfy F. So we have a contradiction. Now, assume that D contains w. Then D is resolvable with C on w and $D \in G$. So D cannot be in $F \setminus G$ and we have a contradiction again.

Proposition 1. *Let $F(X,Y)$ be a formula and $C(X,Y)$ be a clause blocked in F at $w \in X$ with respect to Y. Then $F \ddot{\Rightarrow} C$ with respect to Y.*

Proof. One needs to show that for every full assignment $\vec{y}$ to Y, $(F \wedge C)_{\vec{y}}$ and $F_{\vec{y}}$ are equisatisfiable. If $\vec{y}$ satisfies C, it is trivially true. Assume that $\vec{y}$ does not satisfy C. Since $C_{\vec{y}}$ is blocked in $F_{\vec{y}}$ at w, from Lemma 1 if follows that $F_{\vec{y}} \ddot{\Rightarrow} C_{\vec{y}}$.

Proposition 2. *Let $F(X,Y)$ be a formula and C_{trg} be a clause of F. Let C_{trg} be blocked in F at $w \in X$ with respect to Y in subspace $\vec{q}$ where $w \notin Vars(\vec{q})$. Let $l(w)$ be the literal of w that is in C_{trg}. Let C_{trg} be resolvable with all clauses of F with $\overline{l(w)}$. Let $K = C^* \vee l(w)$ where C^* is the longest clause falsified by $\vec{q}$. Then $K \Rightarrow C_{trg}$ in subspace $\vec{q}$ and $F \setminus \{C_{trg}\} \ddot{\Rightarrow} K$ with respect to Y.*

Proof. Let Q denote $F \setminus \{C_{trg}\}$. One needs to show that for every full assignment $\vec{y}$ to Y, $(K \wedge Q)_{\vec{y}}$ and $Q_{\vec{y}}$ are equisatisfiable. If $\vec{y}$ satisfies K, it is trivially true. Let $\vec{y}$ not satisfy K. Assume the contrary i.e. $(K \wedge Q)_{\vec{y}}$ and $Q_{\vec{y}}$ are not equisatisfiable. Since, $(K \wedge Q)_{\vec{y}} \Rightarrow Q_{\vec{y}}$, the only possibility here is that $Q_{\vec{y}}$ is satisfiable whereas $(K \wedge Q)_{\vec{y}}$ is not. Let $\vec{p}$ be a full assignment to $X \cup Y$ satisfying Q where $\vec{y} \subseteq \vec{p}$. Since $(K \wedge Q)_{\vec{y}}$ is unsatisfiable, $\vec{p}$ falsifies K. This means that $\vec{q} \subseteq \vec{p}$. Recall that the clause C_{trg} is blocked in subspace $\vec{q}$ at w. Using the reasoning of Lemma 1 one can show the existence of $\vec{p'}$ where $\vec{y} \subseteq \vec{p'}$ that satisfies formula Q and the literal $l(w)$. Then $\vec{p'}$ satisfies K and hence $(K \wedge Q)_{\vec{y}}$ is satisfiable. So we have a contradiction.

Remark 2. One can easily extend Proposition 2 to the case where there are clauses of F with literal $\overline{l(w)}$ that are *not* resolvable with C_{trg} on w. Denote the set of such clauses as R. In this case, one may need to add more literals of C_{trg} to K. The objective of adding such literals is to guarantee that K is also unresolvable on w with the clauses of R unsatisfied by $\vec{q}$. Assume that this is the case. Then the proof of Proposition 2 can be reused to show that the extended clause K is es-implied by $F \setminus \{C_{trg}\}$ with respect to Y.

B.2 Special case where es-implication behaves like implication

Let F be a formula and C be a clause of F and K be a clause. If $F \setminus \{C\} \Rightarrow K$, then $F \Rightarrow K$ too. Note, however, that this claim does not extend to es-implication. (That is $F \setminus \{C\} \ddot{\Rightarrow} K$ does not entail $F \ddot{\Rightarrow} K$.) Lemma 2 describes a special case where the claim above holds for es-implication *as well*.

Lemma 2. *Let F, C_{trg} and K be formula and clauses from Proposition 2. (So $F \setminus \{C_{trg}\} \ddot{\Rightarrow} K$ with respect to Y and C_{trg} and K share literal $l(w)$.) Then $F \ddot{\Rightarrow} K$ with respect to Y.*

Proof. Let $\vec{y}$ be a full assignment to Y. Assume the contrary i.e. $(K \wedge F)_{\vec{y}}$ and $F_{\vec{y}}$ are not equisatisfiable. Since $(K \wedge F)_{\vec{y}} \Rightarrow F_{\vec{y}}$, the only possibility here is that $F_{\vec{y}}$ is satisfiable whereas $(K \wedge F)_{\vec{y}}$ is not. Let $\vec{p}$ be a full assignment to $X \cup Y$ satisfying $F_{\vec{y}}$ where $\vec{y} \subseteq \vec{p}$. Assume that $\vec{p}$ satisfies $l(w)$. Then $\vec{p}$ satisfies K and hence $(K \wedge F)_{\vec{y}}$ is satisfiable. So, we have a contradiction.

Now assume that $\vec{p}$ falsifies $l(w)$. Since formula $(K \wedge F)_{\vec{y}}$ is unsatisfiable, $\vec{p}$ falsifies K. Using the reasoning of Lemma 1, one can transform $\vec{p}$ to an assignment $\vec{p'}$ satisfying $(\{K\} \cup (F \setminus \{C_{trg}\}))_{\vec{y}}$ and literal $l(w)$. (The assignment $\vec{p'}$ is obtained from $\vec{p}$ by flipping values assigned to variables of X.) This means that $\vec{p'}$ satisfies C_{trg} too and hence $(K \wedge F)_{\vec{y}}$ is satisfiable. So, we have a contradiction.

C Correctness of *START*

In this appendix, we give a proof that *START* is correct. Let *START* be used to take F_1 out of the scope of quantifiers in $\exists X[F_1(X,Y) \wedge F_2(X,Y)]$. In Subsection C.2, we show that *START* is sound. Then we discuss the following problem. The current implementation of *START* may produce duplicate clauses. In Subsection C.1, we give a simple solution to this problem. In Subsection C.3, we show that the versions of *START* that do not produce duplicate clauses are complete.

C.1 Avoiding generation of duplicate clauses

In this subsection, we make the following two points. First, the *PrvRed* procedure (called in the main loop of *START* to prove redundancy of an X-clause of F_1) cannot generate a duplicate of a clause that is currently in $F_1 \wedge F_2$. Second, *PrvRed can* generate a duplicate of an X-clause of F_1 that was proved redundant earlier and so removed from the formula. This can potentially lead to looping. Fortunately, this problem can be easily fixed by a slight modification of *START*. We did not implement this modification yet since no looping was observed on the formulas we used in experiments of Section 11. (However, we did observe looping on some random formulas we used to verify the soundness of *START*, see Appendix E).

The reason why *PrvRed* does not generate a duplicate of a clause currently present in $F_1 \wedge F_2$ is as follows. First, *PrvRed* does not remove any clauses. Even if the target clause C_{trg} of some recursive call is proved redundant by deriving a certificate, it stays in the formula. (In this case, *PrvRed* simply ignores it in BCP.) Second, *PrvRed* cannot derive a clause C'' implied by a clause C' of $F_1 \wedge F_2$ for the same reason such a derivation cannot be done by a SAT-solver. Namely, an assignment derived from C' by BCP would have satisfied a clause C^* involved in derivation of C'' before C^* became unit. So, derivation of C'' by resolution would not be possible.

Suppose that an X-clause C' of F_1 is proved redundant in the main loop of *START*. Since C' is *removed* from the formula, *START* may actually derive (and add to the formula) a clause C'' that is identical to C'. A simple solution

to this problem is to split C'' on a variable $y_i \in Y$ such that $y_i \notin Vars(C'')$. The idea here is to replace the X-clause C'' with X-clauses $C'' \vee y_i$ and $C'' \vee \overline{y}_i$. So instead of one *old* clause one derives two *new* clauses.

Now suppose that $Y \subset Vars(C'')$ and so there are no variables of Y to split on. Then one can do the following. Let $B_{\vec{y}}$ denote the clause consisting of the literals $l(w)$ of C'' where $w \in Y$. Let $\vec{y}$ be the full assignment to Y falsifying $B_{\vec{y}}$. To prove redundancy of C'' one can run a SAT-solver to check the satisfiability of formula $(F_1 \vee F_2)_{\vec{y}}$. If this formula is unsatisfiable, then $B_{\vec{y}}$ is implied by $F_1 \wedge F_2$ and is a participant-certificate of redundancy of C''. (So, $B_{\vec{y}}$ needs to be a added to the formula.) Let $(F_1 \vee F_2)_{\vec{y}}$ be satisfiable and $\vec{p}$ be a satisfying assignment such that $\vec{y} \subseteq \vec{p}$. Then one can form a witness-certificate K equal to $B_{\vec{y}} \vee l(w)$ were $w \in X$, and $l(w)$ is a literal of C'' satisfied by $\vec{p}$. The clause K obviously implies C'' in subspace $\vec{y}$. Besides, K is es-implied by $F_1 \wedge F_2$ with respect to Y.

C.2 *START* is sound

Our proof of soundness consists of two steps. First, we argue that whenever *START* adds/removes a clause it preserves the equisatisfiability of the current formula $F_1 \wedge F_2$ for every full assignment $\vec{y}$ to Y. Second, we show that the first step guarantees that the solution produced by *START* is correct.

The claim of the first step is based on the following three observations.

- Resolution is sound. So, a clause obtained by resolving clauses of $F_1 \wedge F_2$ is implied by the latter.
- Proposition 2 entails that the es-clause C generated when the current target clause is blocked is es-implied by $F_1 \wedge F_2$. So any clause obtained by resolving C and clauses of $F_1 \wedge F_2$ is es-implied by the latter as well.
- A clause C_{trg} is removed from $F_1 \wedge F_2$ when the former is implied by a certificate K. The clause K is es-implied or implied by $F_1 \wedge F_2$. So removing C_{trg} preserves the equisatisfiability of $F_1 \wedge F_2$.

Let F_1^{ini} and F_2^{ini} denote the *initial* formulas F_1 and F_2. Let $F_1^*(Y)$ consist of the clauses depending only on Y that were present in F_1^{ini} or were generated by *START* by resolution using clauses of F_1^{ini}. The claim of the second step is based on the following two observations.

- Any clause $C(Y)$ es-implied by $F_1 \wedge F_2$ with respect to Y is *implied* by $F_1 \wedge F_2$.
- When *START* terminates, all X-clauses of F_1^{ini} or X-clauses generated using clauses of F_1^{ini} are removed as redundant. So $\exists X[F_1^{ini} \wedge F_2^{ini}] \equiv \exists X[F_1 \wedge F_2] \equiv F_1^* \wedge \exists X[F_2]$. Formula F_2 consists only of clauses of F_2^{ini} and clauses generated by *START* that are es-implied by $F_1 \wedge F_2$. Besides, none of the clauses used to generate new clauses of F_2 has been removed. Hence, these clauses are also es-implied by $F_1^* \wedge F_2^{ini}$. So, $\exists X[F_1^{ini} \wedge F_2^{ini}] \equiv F_1^* \wedge \exists X[F_2^{ini}]$.

C.3 *START* is complete

In this section, we show the completeness of the versions of *START* that do not generate duplicate clauses. (An example of such a version is given in the previous subsection). The completeness of *START* follows from the fact that

- some backtracking condition of *PrvRed* is always met when assigning variables of $X \cup Y$;
- the number of times *START* calls *PrvRed* (to prove redundancy of the current target clause) is finite;
- the number of steps performed by one call of *PrvRed* is finite.

First, let us show that *PrvRed* always meets a backtracking condition. Let $\vec{y}$ be a full assignment to Y. If formula $(F_1 \wedge F_2)_{\vec{y}}$ is unsatisfiable, a clause of $F_1 \wedge F_2$ gets falsified when $\vec{y}$ is extended by assigning the variables of X or earlier. This triggers a backtracking condition.

Now assume that $(F_1 \wedge F_2)_{\vec{y}}$ is satisfiable. Let C_{trg} be the target clause of the last recursive call of *PrvRed*. Let $\vec{p}$ be a full assignment to $X \cup Y$ satisfying $F_1 \wedge F_2$ obtained by extending the assignment $\vec{y}$. Let $l(w)$ be the literal of a variable $w \in Vars(C_{trg}) \cap X$ that is in C_{trg}. Assume that *PrvRed* assigns variables of X as in $\vec{p}$. Suppose that an assignment $w = b$ of $\vec{p}$ satisfies $l(w)$ and hence C_{trg}. First, assume that $w = b$ is derived from a clause C of $F_1 \wedge F_2$ that is currently unit. (Recall that *START* does not make *decision* assignments satisfying C_{trg}.) This means that C implies C_{trg} in the current subspace and a backtracking condition is met. Now, consider the worse case scenario: all the variables of X but w are already assigned and the value of w is not derived from a clause of $F_1 \wedge F_2$. Then C_{trg} is blocked at variable w. Indeed, assume the contrary i.e. there is a clause C that contains the literal $\overline{l(w)}$ and is not satisfied yet. That is all the literals of C other than $\overline{l(w)}$ are falsified by $\vec{p}$. Then $\vec{p}$ cannot be a satisfying assignment because it falsifies either clause C_{trg} or clause C (depending on how the variable w is assigned). So even in the worst case scenario, C_{trg} gets blocked before all variables of $X \cup Y$ are assigned.

Now let us show that *PrvRed* is called a finite number of times. By our assumption, *START* does not generate clauses seen before. So, the number of times *PrvRed* is called in the main loop of *START* (see Figure 1) is finite. *PrvRed* recursively calls itself when the current target clause C_{trg} becomes unit. The number of such calls is finite (since the number of clauses that can be resolved with C_{trg} on its unassigned variable is finite). The clause C_{trg} is satisfied by *PrvRed* before a recursive call. So a clause cannot be used as a target more than once on a path of the search tree. Thus, the number of recursive calls made by *PrvRed* invoked in the main loop of *START* is finite.

When working with a *particular* target clause C_{trg}, *PrvRed* examines a finite search tree. (Here we ignore the steps taken by recursive calls of *PrvRed*). So the number of steps performed by *PrvRed* when operating on C_{trg} is finite.

D Generation Of Circuits Used In Experiments

In this appendix, we explain

- how the circuit N used in experiments is created from the transition relation M of an HWMCC-10 benchmark and
- how the subset X' of input variables X of N used in the experiments is formed.

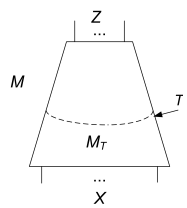

Fig. 5: Cut T and circuit M_T

Let $M(X, Y, Z)$ be the circuit specifying the transition relation of an HWMCC-10 benchmark. Here X, Y, Z specify the input, internal and output variables of M. Let $F(X, Y, Z)$ be a formula specifying the circuit M. Checking if an input constraint $G(X)$ preserves the range of M is an instance of PQE. Namely, it reduces to verifying the redundancy of G in $\exists W[G \wedge F]$ where $W = X \cup Y$.

To decrease the size of the circuit to consider, we used the following idea. Let T be a set of gates forming a cut in M (see Fig. 5). Let $M_T(X, Y_T, Z_T)$ be the subcircuit of M located below T. A nice property of the circuit M_T is that if $G(X)$ preserves the range of M_T, it does the same for M (although the opposite may not be true). Thus, instead of checking redundancy of G in $\exists W[G \wedge F]$, one can check that of G in $\exists W_T[G \wedge F_T]$. Here $W_T = X \cup Y_T$ and $F_T(X, Y_T, Z_T)$ is a formula specifying M_T. So, in the experiments we used the circuit M_T instead of M i.e. M_T is the circuit N mentioned in Section 11.

In experiments, we formed the cut T of M from gates with topological levels[15] less or equal to k. Suppose that an input $x \in X$ feeds a gate G whose topological level is greater than k (see Fig. 6). We will refer to x as a *cut input*. In this case, a buffer B whose input is fed by x is added to M_T. The output of B becomes a new output of M_T. We will refer to $x \in X$ as a *non-cut input* if it does not feed a gate with topological level greater than k. To form the cut T of M we picked the smallest value of $k \geq 5$ for which at least one of the two conditions held

- the number of non-cut inputs was greater than 50
- at least 5% of inputs were non-cut ones.

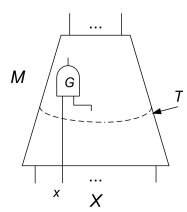

Fig. 6: Cut input x

We took into consideration only non-cut inputs because constraining a cut input x obviously changed the range of M_T (due to the buffer B fed by x). So, when picking the subset of X' of inputs to constrain we used only non-cut ones. Namely, X' consisted of all non-cut input variables if their number was less or equal to 50. Otherwise, we formed X' by picking the first 50 non-cut input variables listed in the description of the circuit M.

Table 4 shows statistics for some circuits M_T used in the experiments. For every circuit of Table 4, *START* solved at least one of $2 * |X'|$ PQE problems with the time limit of 1 sec. The

[15] The topological level of a gate of circuit M is defined recursively as follows. The topological level of an input of M is 0. The topological level of a gate G of M is equal to the maximum level among the fanin gates of G plus 1.

first column gives the name of the HWMCC-10 benchmark specifying a transition relation M. The following three columns show the size of M_T in terms of the number of gates, inputs and outputs. The last column gives the number of topological levels in M_T (i.e. the value k above). To give a more realistic picture of the size of circuits M_T, Table 4 does not include the trivial logic introduced via buffers B. Namely, the number of gates of M_T given in Table 4 does not include buffers B. The number of inputs of M_T does not include those that feed only buffers B. The number of outputs of M_T does not include those introduced via adding buffers B.

E Verifying A Solution To PQE By A Sat-Solver

Table 4: A sample of circuits M_T

name	#gates	#inps	#outs	#lvls
bobsmmem	8,340	2,538	2,366	5
bobtuttt	7,146	2,890	1,000	5
bobsynth00	6,817	2,658	2,617	5
pdtvissfeistel	5,942	429	1,335	8
bobpcihm	4,968	1,300	1,516	5
139443p24	3,599	531	1,089	5
pdtpmsvsa16a	3,131	376	1,053	5
neclaftp4001	2,530	917	1,788	10
pdtswvqis8x8p0	2,240	109	1,014	13
pdtvisbakery3	1,826	39	400	18
pdtvisblackjack4	1,689	108	632	14

To increase confidence in the correctness of our implementation of *START*, we verified its solutions on 500,000 PQE problems generated randomly. We used formulas large enough to exercises every part of the *START* code and small enough for a SAT-based verifier to check the solution produced by *START*. (We will refer to this verifier as *VerSol*.) The size of the formulas ranged from 30 to 50 variables and from 60 to 100 clauses.

Let $F_1^*(Y)$ be a solution to the problem of taking F_1 out of the scope of quantifiers in $\exists X[F_1(X,Y) \wedge F_2(X,Y)]$ i.e. $F_1^* \wedge \exists X[F_2] \equiv \exists X[F_1 \wedge F_2]$. First, for every clause $C \in F_1^*$, *VerSol* checks if $F_1 \wedge F_2 \Rightarrow C$. This reduces to testing the satisfiability of $F_1 \wedge F_1 \wedge \overline{C}$. Then *VerSol* checks redundancy of X-clauses of F_1 one by one. Namely, given an X-clause $C \in F_1$, *VerSol* checks if C is redundant in $\exists X[F_1^* \wedge F_1 \wedge F_2]$. If not, then the solution F_1^* is incorrect. Otherwise, C is removed from F_1 and a new X-clause of F_1 is picked (unless every X-clause is already removed from F_1).

Checking redundancy of $C \in F_1$ by a SAT-solver is based on the following observation. Let $\vec{p}$ be a full assignment to $X \cup Y$ satisfying $F_1^* \cup (F_1 \setminus \{C\}) \cup F_2$ and falsifying C. Let $\vec{y}$ be a full assignment to Y such that $\vec{y} \subseteq \vec{p}$. If $(F_1^* \wedge F_1 \wedge F_2)_{\vec{y}}$ is unsatisfiable, then C is *not* redundant in $\exists X[F_1^* \wedge F_1 \wedge F_2]$ (because $(F_1^* \wedge F_1 \wedge F_2)_{\vec{y}}$ becomes satisfiable after removing C). So verification of redundancy of C reduces to enumerating assignments $\vec{p}$ satisfying $F_1^* \wedge G_1 \wedge F_2 \wedge \overline{C}$ where $G_1 = F_1 \setminus \{C\}$ and checking if $(F_1^* \wedge F_1 \wedge F_2)_{\vec{y}}$ is satisfiable where $\vec{y} \subseteq \vec{p}$ and $Vars(\vec{y}) = Y$.