Partial Quantifier Elimination By Certificate Clauses

Eugene Goldberg

eu.goldberg@gmail.com

Abstract. We study partial quantifier elimination (PQE) for propositional CNF formulas. PQE is a generalization of quantifier elimination where one can limit the set of clauses taken out of the scope of quantifiers to a small subset of target clauses. The appeal of PQE is twofold. First, PQE can be dramatically simpler than full quantifier elimination. Second, PQE provides a language for performing incremental computations. Many verification problems (e.g. equivalence checking and model checking) are inherently incremental and so can be solved in terms of PQE. Our approach is based on deriving clauses depending only on unquantified variables that make the target clauses *redundant*. Proving redundancy of a target clause is done by construction of a "certificate" clause implying the former. We describe a PQE algorithm called $START$ that employs the approach above. To evaluate $START$, we apply it to invariant generation for a sequential circuit N. The goal of invariant generation is to find an *unwanted* invariant of N proving unreachability of a state that is supposed to be reachable. If N has an unwanted invariant, it is buggy. Our experiments with FIFO buffers and HWMCC-13 benchmarks suggest that $START$ can be used for detecting bugs that are hard to find by existing methods.

1 Introduction

In this paper, we consider the following problem. Let $F_1(X,Y), F_2(X,Y)$ be propositional formulas in conjunctive normal form (CNF)[1] where X, Y are sets of variables. Given $\exists X[F_1 \wedge F_2]$, find a quantifier-free formula $F_1^*(Y)$ such that $\exists X[F_1 \wedge F_2] \equiv F_1^* \wedge \exists X[F_2]$. In contrast to quantifier elimination (QE), only a part of the formula gets "unquantified" here. So, this problem is called *partial* QE (PQE) [1,2]. We will refer to F_1^* as a *solution* to PQE. Like SAT, PQE is a way to cope with the complexity of QE. But in contrast to SAT that is a *special* case of QE (where all variables are quantified), PQE *generalizes* QE. The latter is just a special case of PQE where $F_2 = \emptyset$ and the entire formula is unquantified. Interpolation [3,4] is also a special case of PQE [5].

The appeal of PQE is twofold. First, it can be much more efficient than QE if F_1 is a *small* part of the formula. Second, PQE provides a language for

[1] Every formula is a propositional CNF formula unless otherwise stated. Given a CNF formula F represented as the conjunction of clauses $C_1 \wedge \cdots \wedge C_k$, we will also consider F as the *set* of clauses $\{C_1, \ldots, C_k\}$.

performing *incremental* computing. So, PQE facilitates the development of new approaches to various verification problems like SAT [6,1], equivalence checking [7], model checking [8] and so on.

We solve PQE by *redundancy based reasoning.* Its introduction is motivated by the following observations. First, $F_1 \wedge F_2 \Rightarrow F_1^*$ and $F_1^* \wedge \exists X[F_1 \wedge F_2] \equiv F_1^* \wedge \exists X[F_2]$. 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.* Second, one can prove clauses of F_1 redundant[2] one by one. The redundancy of a clause $C \in F_1$ can be proved by using $(F_1 \cup F_2) \setminus \{C\}$ to derive a clause K implying C. We refer to K as a *certificate clause.* Importantly, one can produce K even if $(F_1 \cup F_2) \setminus \{C\}$ does not imply C. This becomes possible if one allows generation of clauses preserving *equisatisfiability* rather than equivalence.

We implement redundancy based reasoning in a PQE algorithm called $START$, an abbreviation of Single TARgeT. At any given moment, $START$ proves redundancy of only one clause (hence the name "single target"). $START$ builds the certificate K above by resolving "local" certificate clauses implying the clause C in subspaces. Proving redundancy of C in subspaces where $F_1 \wedge F_2$ is unsatisfiable, in general, requires adding new clauses to $F_1 \wedge F_2$. The added clauses depending only on unquantified variables form a solution F_1^* to the PQE problem. $START$ is somewhat similar to a SAT-solver with conflict driven learning. A major difference here is that $START$ backtracks as soon as the target clause is proved redundant in the current subspace (even if no conflict occurred).

The main body of this paper is structured as follows. (Some additional information is provided in the appendix.) Section 3 shows that interpolation is a special case of PQE. In Section 4, to demonstrate the versatility of PQE, we recall reductions of a few well known problems to PQE. A description of $START$ is given in Sections 6-8. Ideally, we would like to apply $START$ to a known problem e.g. one listed in Section 4 and compare it with existing tools. However, PQE-solving is still in its infancy and must go through many improvements before it matures. (PQE is much more complex than, say, SAT. As mentioned above, SAT is a special case of QE and QE is a special case of PQE.) So instead, in Sections 9-10, we apply $START$ to the problem of invariant generation that has no general solution yet. We use invariant generation for bug detection as described below. Our objective here is to provide a "proof of concept" for PQE i.e. to give some experimental evidence that PQE is an important direction for research.

Let N be a sequential circuit to verify. As far as reachable states of N are concerned, one can have bugs of two kinds. A bug of the first kind occurs if a bad state is reachable in N. A bug of the second kind takes place if a required good state (i.e. one that is supposed to be reachable) is *unreachable* in N. One excludes bugs of the first kind by checking that a set of desired invariants holds. The challenge here is that these invariants may be hard to prove. Bugs of the second kind are currently identified either by testing or by checking if N has an

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

unwanted invariant. An invariant P of N is unwanted if a required good state falsifies P and so is unreachable in N. If P holds for N, the latter has a bug of the second kind. The unwanted invariants to check are currently generated manually i.e. are guessed. So, one can easily overlook a bug of the second kind. The main challenge here is to *find* an unwanted invariant that holds rather than the hardness of proving it true. In Section 9, we show that PQE can be used to *automatically* generate invariants to check for being unwanted. In Section 10, we use *START* to detect a bug of the second kind in a FIFO buffer that is hard to find by existing methods. In Appendix F, we present results showing that *START* is efficient enough to generate invariants of HWMCC-13 benchmarks. We also give evidence that PQE can be dramatically more efficient than QE. Finally, Section 11 explains how to decide if an invariant is unwanted via test generation.

2 Basic Definitions

We assume that every formula is in CNF unless otherwise stated. In this section, when we say "formula" without mentioning quantifiers, we mean "a quantifier-free formula".

Definition 1. *Let F be a formula. Then* $\boldsymbol{Vars(F)}$ *denotes the set of variables of F and* $\boldsymbol{Vars(\exists X[F])}$ *denotes* $Vars(F) \setminus X$.

Definition 2. *Let V be a set of variables. An* ***assignment*** *$\vec{q}$ to V is a mapping $V' \rightarrow \{0,1\}$ where $V' \subseteq V$. We will denote the set of variables assigned in $\vec{q}$ as* $\boldsymbol{Vars(\vec{q})}$*. We will refer to $\vec{q}$ as a* ***full assignment*** *to V if $Vars(\vec{q}) = V$. 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 3. *Let C be a* ***clause*** *(i.e. a disjunction of literals). Let H be a formula that may have quantifiers, and $\vec{q}$ be an assignment to $Vars(H)$. If C is satisfied by $\vec{q}$, then* $\boldsymbol{C_{\vec{q}} \equiv 1}$*. Otherwise,* $\boldsymbol{C_{\vec{q}}}$ *is the clause obtained from C by removing all literals falsified by $\vec{q}$. Denote by* $\boldsymbol{H_{\vec{q}}}$ *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 4. *Given a formula $\exists X[F(X,Y)]$, a clause C of F is called a* ***quantified clause*** *if $Vars(C) \cap X \neq \emptyset$. If $Vars(C) \cap X = \emptyset$, the clause C depends only on free i.e. unquantified variables of F and is called a* ***free clause****.*

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

Definition 6. *Let F be a formula and $G \subseteq F$ and $G \neq \emptyset$. Formula G is* ***redundant in*** $\boldsymbol{\exists X[F]}$ *if $\exists X[F] \equiv \exists X[F \setminus G]$.*

Definition 7. *Given a formula $\exists X[F_1(X,Y) \wedge F_2(X,Y)]$, the **Partial Quantifier Elimination (PQE)** problem is to find $F_1^*(Y)$ such that $\exists \boldsymbol{X}[\boldsymbol{F_1} \wedge \boldsymbol{F_2}] \equiv \boldsymbol{F_1^*} \wedge \exists \boldsymbol{X}[\boldsymbol{F_2}]$. (So, PQE takes F_1 out of the scope of quantifiers.) F_1^* is called a **solution** to PQE. The case of PQE where $F_2 = \emptyset$ is called **Quantifier Elimination (QE)**.*

Remark 1. Let C be a clause of a solution F_1^* to the PQE problem above. If F_2 implies C, then $F_1^* \setminus \{C\}$ is a solution too.

3 PQE And Interpolation

In this section, we recall the observation of [5] that interpolation is a special case of PQE. Let $A(X,Y) \wedge B(Y,Z)$ be an unsatisfiable formula. Let $I(Y)$ be a formula such that $A \wedge B \equiv I \wedge B$ and $A \Rightarrow I$. Then I is called an *interpolant* [3]. Now, let us show that interpolation can be described in terms of PQE. Consider the formula $\exists W[A \wedge B]$ where A and B are the formulas above and $W = X \cup Z$. Let $A^*(Y)$ be obtained by taking A out of the scope of quantifiers i.e. $\exists W[A \wedge B] \equiv A^* \wedge \exists W[B]$. Since $A \wedge B$ is unsatisfiable, $A^* \wedge B$ is unsatisfiable too. So, $A \wedge B \equiv A^* \wedge B$. If $A \Rightarrow A^*$, then A^* is an interpolant.

The *general case* of PQE that takes A out of $\exists W[A \wedge B]$ is different from the instance above in three aspects. First, one does not assume that $A \wedge B$ is unsatisfiable. Second, one does not assume that $\mathit{Vars}(B) \subset \mathit{Vars}(A \wedge B)$. In other words, in general, PQE *does not* remove any variables from the original formula. Third, a solution A^* is implied by $A \wedge B$ rather than by A alone. Summarizing, one can say that interpolation is a special case of PQE.

4 Examples Of Problems That Reduce To PQE

In this section, we give a few examples of how a problem can be reduced to PQE. In Section 9, we show how one can use PQE to generate invariants.

4.1 SAT-solving by PQE [1]

Consider the SAT problem of checking if formula $\exists X[F(X)]$ is true. One can view traditional SAT-solving as proving *all* clauses redundant in $\exists X[F]$ e.g. by finding a satisfying assignment or by deriving an empty clause and adding it to F. The reduction to PQE below facilitates developing an incremental SAT-algorithm that needs to prove redundancy only for a *fraction* of clauses.

Let $\vec{x}$ be a full assignment to X and H denote the clauses of F falsified by $\vec{x}$. Checking the satisfiability of F reduces to taking H out of the scope of quantifiers i.e. to finding H^* such that $\exists X[F] \equiv H^* \wedge \exists X[F \setminus H]$. Since all variables of F are quantified in $\exists X[F]$, the formula H^* is a Boolean constant 0 or 1. If $H^*=0$, F is unsatisfiable. If $H^*=1$, then F is satisfiable because $F \setminus H$ is satisfied by $\vec{x}$.

4.2 Equivalence checking by PQE [7]

Let $N'(X',Y',z')$ and $N''(X'',Y'',z'')$ be single-output combinational circuits to check for equivalence. Here X^*,Y^* are the sets of input and internal variables and z^* is the output variable of N^*. The reduction to PQE below facilitates the design of a *complete* algorithm able to exploit the similarity of N' and N''. This is important because the current equivalence checkers exploiting such similarity are *incomplete*. If N' and N'' are not "similar enough", e.g. they have no functionally equivalent internal points, the equivalence checker invokes a complete (but inefficient) procedure that ignores similarity of N' and N''.

Let $eq(X',X'')$ specify a formula such that $eq(\vec{x}\,',\vec{x}\,'') = 1$ iff $\vec{x}\,' = \vec{x}\,''$ where $\vec{x}\,',\vec{x}\,''$ are full assignments to X' and X''. Let formulas $G'(X',Y',z')$ and $G''(X'',Y'',z'')$ specify N' and N'' respectively. (As usual, we assume that a formula G specifying a circuit N is obtained by Tseitin transformations [9].) Let $h(z',z'')$ be a formula obtained by taking eq out of $\exists W[eq \wedge G' \wedge G'']$ where $W = X' \cup Y' \cup X'' \cup Y''$. That is $\exists W[eq \wedge G' \wedge G''] \equiv h \wedge \exists W[G' \wedge G'']$. If $h \Rightarrow (z' \equiv z'')$, then N' and N'' are equivalent. Otherwise, N' and N'' are inequivalent, unless they are identical constants i.e. $z' \equiv z'' \equiv 1$ or $z' \equiv z'' \equiv 0$. It is formally proved in [7] that the more similar N', N'' are (where similarity is defined in the most general sense), the easier taking eq out of $\exists W[eq \wedge G' \wedge G'']$ becomes.

4.3 Computing reachability diameter by PQE [8]

One can use PQE to find the reachability diameter of a transition system without computing the set of all reachable states. So, one can prove an invariant by PQE without generating a stronger invariant that is inductive.

Let formulas $T(S_j,S_{j+1})$ and $I(S_0)$ specify the transition relation and initial states of a transition system ξ. Here S_j denotes the set of state variables of j-th time frame. For the sake of simplicity, we assume that ξ is able to stutter i.e. $T(\vec{s},\vec{s}) = 1$, for every state $\vec{s}$. (Then the sets of states reachable in m transitions and *at most* m transitions are identical. If T does not have the stuttering feature it can be easily introduced.)

Let $Diam(I,T)$ denote the *reachability diameter* for initial states I and transition relation T. That is every state of the system ξ can be reached in at most $Diam(I,T)$ transitions. Given a number m, one can use PQE to decide if $Diam(I,T) < m$. This is done by checking if I_1 is redundant in $\exists \mathbb{S}_{m-1}[I_0 \wedge I_1 \wedge \mathbb{T}_m]$. Here I_0 and I_1 are initial states in terms of variables of S_0 and S_1 respectively, $\mathbb{S}_{m-1} = S_0 \cup \dots \cup S_{m-1}$ and $\mathbb{T}_m = T(S_0,S_1) \wedge \dots \wedge T(S_{m-1},S_m)$. If I_1 is redundant, then $Diam(I,T) < m$ holds.

The idea above can be used, for instance, to prove an invariant P true in an IC3-like manner (i.e. by constraining P) but without turning P into an inductive invariant. To prove P true, it suffices to constrain P to a formula H such that a) $I \Rightarrow H \Rightarrow P$, b) $Diam(H,T) < m$ and c) no state falsifying P can be reached from a state satisfying H in $m-1$ transitions. The conditions b) and c) can be verified by PQE and bounded model checking [10] respectively. In the special case of H meeting the three conditions above for $m = 1$, H is an *inductive invariant.*

5 Extended Implication And Blocked Clauses

One can introduce the notion of implication via that of redundancy. Namely, $F \Rightarrow G$, iff G is redundant in $F \wedge G$ i.e. iff $F \wedge G \equiv F$. We use this idea to extend the notion of implication via redundancy in a *quantified* formula.

Definition 8. *Let $F(X,Y)$ and $G(X,Y)$ be formulas and G be redundant in $\exists X[F \wedge G]$ i.e. $\exists X[F \wedge G] \equiv \exists X[F]$. Then $(F \wedge G)_{\vec{y}}$ and $F_{\vec{y}}$ are equisatisfiable for every full assignment $\vec{y}$ to Y. So, we will say that F* ***es-implies*** *G in $\exists X[F \wedge G]$. (Here "es" stands for "equisatisfiability".) A clause C is called an* ***es-clause*** *in $\exists X[F \wedge C]$ if F es-implies C in $\exists X[F \wedge C]$. One can view es-implication as a weaker version of regular implication.*

Note that if F implies G, then F also es-implies G in $\exists X[F \wedge G]$. However, the converse is not true. We will say that F es-implies G without mentioning the formula $\exists X[F \wedge G]$ if the latter is clear from the context.

Definition 9. *Let clauses C',C'' have opposite literals of exactly one variable $w \in \mathit{Vars}(C') \cap \mathit{Vars}(C'')$. Then 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''. The clause C is said to be obtained by* ***resolution*** *on w.*

Clauses C', C'' having opposite literals of more than one variable are considered **unresolvable** to avoid producing a tautologous resolvent C (i.e. $C \equiv 1$).

Definition 10. *Given a formula $\exists X[F(X,Y)]$, let C be a clause of F. Let G be the set of clauses of F resolvable with C on a variable $w \in X$. Let $w = b$ satisfy C, where $b \in \{0,1\}$. We will call C* ***blocked*** *in $\exists X[F]$ at w if G is redundant in $\exists X[F]$ in subspace $w = b$ (i.e. if $G_{w=b}$ is redundant in $\exists X[F_{w=b}]$).*

Remark 2. Note that if $G = \emptyset$ or the clauses of G are removed from $\exists X[F]$ as redundant, C meets the *original* definition of a blocked clause [11]. Definition 10 allows to declare C blocked *without* removing clauses of G if a proof of their redundancy in $\exists X[F]$ is available. This feature is used by our PQE-solver *START* (see Remark 4 of Section 8).

Proposition 1. *Given a formula $\exists X[F(X,Y)]$, let C be a clause blocked in $\exists X[F]$ at $w \in X$. Then C is redundant in $\exists X[F]$ i.e. $\exists X[F] \equiv \exists X[F \setminus \{C\}]$. So, C is es-implied by $F \setminus \{C\}$ in $\exists X[F]$.*

Proofs of the propositions are given in Appendix A.

6 A Simple Example Of How *START* Operates

In this paper, we introduce a PQE algorithm called *START* (an abbreviation of Single TARgeT). In this section, we give a taste of *START* by a simple example. Figure 1 describes how *START* operates on the problem shown in lines 1-6. (Figure 1 and Figures 6,7,8 of the appendix are built using a version of *START* generating execution traces. A Linux binary of this version can be downloaded from [12].)

```
1   Find $F_1^*(Y)$ such that
2   $\exists X[F_1 \wedge F_2] \equiv F_1^* \wedge \exists X[F_2]$
3   $Y = \{y_1\}$, $X = \{x_2, x_3\}$
4   $F_1 = \{C_1\}$, $C_1 = \overline{x}_2 \vee x_3$
5   $F_2 = \{C_2, C_3\}$, $C_2 = y_1 \vee x_2$,
6       $C_3 = y_1 \vee \overline{x}_3$
7   pick. $C_1 \in F_1$ to prove red.
8   — call $PrvRed$—
9   decision: $y_1 = 0$ at level 1
10  $BCP$:$(C_2 : x_2 = 1)(C_3 : x_3 = 0)$
11  LEAF: conflict at level 1
12  $C_1 = \overline{x}_2 \vee x_3$ is falsified
13  gen. particip. cert. $C_4 = y_1$
14      $R_1 = Res(C_1, C_2, x_2)$,
15      $C_4 = Res(R_1, C_3, x_3)$
16  $F_1 = F_1 \cup \{C_4\}$
17  backtracking to level 0
18  $BCP$: $(C_4 : y_1 = 1)$
19  LEAF: $C_1$ is blocked at $x_2$
20  (since $C_2$ is sat. by $y_1 = 1$)
21  $K_1 = \overline{y}_1 \vee \overline{x}_2$ is the init. cert.
22  $K_2 = \overline{x}_2$ is the final cert.
23      $K_2 = Res(K_1, C_4, y_1)$
24  $K_1, K_2$ are witness certs.
25  not added to $F_1 \wedge F_2$
26  — exit $PrvRed$—
27  $K_2$ is a global certif.
28  $F_1 := F_1 \setminus \{C_1\}$
29  Sol. $F_1^* = F_1 = \{C_4\}$
```

Fig. 1: *START*, an example of operation

First, *START* picks C_1, the only quantified clause of F_1. We will refer to C_1 as the **target clause**. Then *START* invokes a procedure called *PrvRed* to prove C_1 redundant (lines 8-26). The algorithm of *PrvRed* is somewhat similar to that of a SAT-solver [13]. *PrvRed* makes decision assignments and runs *BCP* (Boolean Constraint Propagation). Besides, *PrvRed* uses the notion of a **decision level** that consists of a decision assignment and implied assignments derived by *BCP*. (The decision level number 0 is an exception. It has only implied assignments.) On the other hand, there are a few important differences. In particular, *PrvRed* has a richer set of backtracking conditions, a conflict being just one of them.

PrvRed starts the decision level number 1 by making assignment $y_1 = 0$. Then it runs *BCP* to derive assignments $x_2 = 1$ and $x_3 = 0$ from clauses C_2 and C_3 that became **unit** (i.e. have only one unassigned variable). At this point, a conflict occurs since C_1 is falsified (lines 11-16). Then *PrvRed* generates conflict clause $C_4 = y_1$. It is built like a regular conflict clause [13]. Namely, C_4 is obtained by resolving C_1 with C_2 and C_3 to eliminate the variables whose values were derived by *BCP* at decision level 1. The clause C_4 *certifies* that C_1 is redundant in $\exists X[F_1 \wedge F_2]$ in subspace $y_1 = 0$. We call a clause like C_4 **a certificate**. Note that C_1 becomes redundant only after *adding* C_4 to the formula, because C_1 itself is involved in the derivation of C_4. We will refer to the certificates one has to add to the formula as **participant certificates**. The participant certificates depending only on free variables form a solution to the PQE problem.

After generating C_4, like a SAT-solver, *PrvRed* backtracks to the smallest decision level where C_4 is unit (i.e. level 0) and derives the assignment $y_1 = 1$.

Then the target C_1 is blocked at variable x_2 (lines 19-25). The reason is that C_2, the only clause resolvable with C_1 on x_2, is satisfied by $y_1 = 1$. At this point, *PrvRed* generates the clause $K_1 = \overline{y}_1 \vee \overline{x}_2$. It implies C_1 in subspace $y_1 = 1$, thus certifying its redundancy there. (The construction of K_1 is explained in Example 1 of Subsection 7.3. Importantly, the target C_1 *is not used* in generation of K_1.) By resolving K_1 and $C_4 = y_1$, *PrvRed* builds the final certificate $K_2 = \overline{x}_2$ for the decision level 0. *PrvRed* derives K_2 from K_1 like a SAT-solver derives a conflict clause from a clause falsified at a conflict level. That is K_2 is built by resolving out variables of K_1 assigned by values derived at the current decision level. In our case, it is the variable y_1. Since K_1 and K_2 are derived without using the target clause C_1, one *does not have to* add them to the formula. They just "witness" the redundancy of C_1. We will refer to them as **witness certificates**.

K_2 implies C_1 in the entire space and thus is a global certificate. So, *START* removes C_1 from F_1 (line 28). Since now F_1 does not have quantified clauses, *START* terminates. It returns the current $F_1 = \{C_4\}$ as a solution $F_1^*(Y)$ to the PQE problem. That is $\exists X[C_1 \wedge F_2] \equiv C_4 \wedge \exists X[F_2]$.

7 Description Of *START*

In this section, we describe *START* in more detail. A proof of correctness of *START* is given in Appendix E. For the sake of simplicity, in the current version of *START*, the witness certificates are not added to the formula and so are not reused[3].

7.1 The main loop of *START*

```
START(F_1, F_2, Y){
1 while (true) {
2   $C_{trg}$ := PickQntCls($F_1$)
3   if ($C_{trg}$ = nil) {
4     $F_1^*$ := $F_1$
5     return($F_1^*$)}
6   $\vec{q}$ := $\emptyset$
7   $K$ := PrvRed($F_1 \wedge F_2$, $C_{trg}$, $Y$, $\vec{q}$)
8   if (EmptyCls($K$)) return($K$)
9   $F_1$ := $F_1 \setminus \{C_{trg}\}$}}
```

Fig. 2: The main loop

The main loop of *START* is shown in Fig. 2. *START* accepts formulas $F_1(X,Y)$, $F_2(X,Y)$ and set Y and outputs formula $F_1^*(Y)$ such that $\exists X[F_1 \wedge F_2] \equiv F_1^* \wedge \exists X[F_2]$. The loop begins with picking a quantified clause $C_{trg} \in F_1$ that is the *target* clause to be proved redundant (line 2). If F_1 has no quantified clauses, it is the solution $F_1^*(Y)$ returned by *START* (lines 3-5). Otherwise, *START* initializes the assignment $\vec{q}$ to $X \cup Y$ and invokes a procedure called *PrvRed* to prove C_{trg} redundant (lines 6-7). *PrvRed* returns a clause K implying C_{trg} and thus *certifying* its redundancy. If K is an empty clause (i.e. has no literals), F is unsatisfiable. Then *PrvRed* returns K as a solution to the PQE problem (line 8). Otherwise, K consists of (some) literals of C_{trg}. Besides, K is redundant in $\exists X[K \wedge (F_1 \cup F_2 \setminus \{C_{trg}\})]$.

[3] In practice, witness certificates are derived in subspaces where the formula is satisfiable. So, reusing them should boost the pruning power of *START* in those subspaces.

So, C_{trg} is redundant in $\exists X[C_{trg} \wedge (F_1 \cup F_2 \setminus \{C_{trg}\})]$ and $START$ removes it from F_1 (line 9). In the process of deriving the certificate K above, *PrvRed* may add participant certificates to F_1. If an added certificate clause K' is quantified, *PrvRed* will be called at a later iteration of the main loop to prove K' redundant.

7.2 Description of *PrvRed*

The pseudo-code of *PrvRed* is shown in Fig 3. Let F denote $F_1 \wedge F_2$. The objective of *PrvRed* is to prove the current target clause C_{trg} redundant in $\exists X[F]$ in the subspace specified by an assignment $\vec{q}$ to $X \cup Y$. The reason why one needs $\vec{q}$ is that *PrvRed* can be called recursively in *subspaces* to prove redundancy of some "local" target clauses (Section 8).

First, in line 1, *PrvRed* stores the initial value of $\vec{q}$. (It is used in line 10 to limit the backtracking of *PrvRed*.) Besides, *PrvRed* initializes the assignment queue Q. The main work is done in a loop similar to that of a SAT-solver [13]. The operation of *PrvRed* in this loop is partitioned into two parts separated by the dotted line.

```
// $F$ denotes $F_1 \wedge F_2$
//
$PrvRed(F, C_{trg}, Y, \vec{q})\{$
1  $\vec{q}_{init} := \vec{q}$; $Q = \emptyset$
2  while ($true$) {
3    if ($Q = \emptyset$) {
4      $(v,b) := MakeDec(F, Y, C_{trg})$
5      $UpdQueue(Q, v, b)$ }
6    $K_{bct} := BCP(Q, \vec{q}, F, Y, C_{trg})$
7    if ($K_{bct} = nil$) continue
   − − − − −
8    $K := Lrn(F, \vec{q}, K_{bct})$
9    if ($Particip(K)$) $F_1 := F_1 \cup \{K\}$
10   $Backtrack(\vec{q}_{init}, \vec{q}, K)$
11   if ($\vec{q} = \vec{q}_{init}$) return($K$)
12   $UpdQueue(Q, \vec{q}, K)$}}
```

Fig. 3: The *PrvRed* procedure

The first part (lines 3-7) starts with checking if the assignment queue Q is empty. If so, a decision assignment $v = b$ is picked and added to Q (lines 4-5). Here $v \in (X \cup Y)$ and $b \in \{0,1\}$. The variables of Y are the first to be assigned by *PrvRed*[4]. 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 subspaces where it is satisfied.)

Then *PrvRed* calls the *BCP* procedure. If *BCP* identifies a backtracking condition, it returns a certificate clause K_{bct} implying C_{trg} in the current subspace. (Here, "*bct*" stands for "backtracking" because K_{bct} is the reason for backtracking.) After *BCP*, *PrvRed* goes to the second part of the loop where the actual backtracking is done. If no backtracking condition is met, a new iteration begins.

The certificate K_{bct} returned by *BCP* depends on the backtracking condition. *BCP* identifies three of them: a) a conflict, b) C_{trg} is implied in subspace $\vec{q}$ by an existing clause, and c) C_{trg} is blocked in subspace $\vec{q}$. In the first case, K_{bct} is a clause falsified in the current subspace $\vec{q}$ i.e. one reached during BCP. In the

[4] The goal of $START$ is to derive free clauses making the quantified clauses of F_1 redundant in $\exists X[F_1 \wedge F_2]$. Assigning variables of X after those of Y guarantees that, when generating a new clause, the variables of X are resolved out *before* those of Y.

second case, K_{bct} is a clause that *BCP* made unit and that shares its only literal with C_{trg}. (Such a clause implies C_{trg} in the current subspace $\vec{q}$.) In the third case, K_{bct} is generated by *PrvRed* as described in the next subsection.

PrvRed starts the second part (lines 8-12) with a procedure called *Lrn* that uses K_{bct} to build another certificate K implying C_{trg} in subspace $\vec{q}$. Generation of K from K_{bct} is similar to how a SAT-solver generates a conflict clause from a falsified clause [13]. Namely, when building K, *Lrn* resolves out the variables whose value was *derived* at the decision level where the backtracking condition occurred. If C_{trg} was used to generate K i.e. the latter is a *participant certificate*, K is added to F_1 (line 9). This guarantees that *PrvRed* adds only clauses *implied* by the current formula. (The only es-clauses generated by *PrvRed* and described in the next subsection are used solely to generate *witness certificates*. So, a witness certificate K is, in general, es-implied rather than implied by the formula F in $\exists X[K \wedge F]$. For that reason, in the current version of *START*, witness certificates are not added to the formula. In one special case, to avoid adding a witness certificate, *PrvRed* has to derive and add to the formula a special clause. This case is described in Appendix D.)

After generating K, *PrvRed* backtracks (line 10). The assignment $\vec{q}_{init}$ sets the limit of backtracking. If *PrvRed* reaches this limit, C_{trg} is proved redundant in the required subspace and *PrvRed* terminates (line 11). Otherwise, an assignment is derived from K and added to the queue Q (line 12). This is similar to the backtracking of a SAT-solver to the smallest decision level where the last conflict clause is unit. So, an assignment can be derived from this clause by *BCP*. More information can be found in Appendix B.

7.3 Generation of clause K_{bct} when C_{trg} is blocked

Let C_{trg} get blocked in $\exists X[F]$ in the current subspace $\vec{q}$ during *BCP*. So, C_{trg} is redundant in $\exists X[F]$ in this subspace. Then a clause K_{bct} is generated as described in Proposition 2 where $(K_{bct})_{\vec{q}} \Rightarrow (C_{trg})_{\vec{q}}$ and K_{bct} is redundant in $\exists X[K_{bct} \wedge (F \setminus \{C_{trg}\})]$. Thus, K_{bct} certifies redundancy of C_{trg} in subspace $\vec{q}$ and is returned by *BCP* as the reason for backtracking (line 6 of Fig 3). This is the only case of backtracking where the clause K_{bct} returned by BCP is *es-implied* rather than implied by F in $\exists X[K_{bct} \wedge F]$.

Proposition 2. *Given a formula $\exists X[F(X,Y)]$, let $C_{trg} \in F$. Let $\vec{q}$ be an assignment to $X \cup Y$ that does not satisfy C_{trg}. Let C_{trg} be blocked in $\exists X[F]$ at $w \in X$ in subspace $\vec{q}$ where $w \notin Vars(\vec{q})$. Let $l(w)$ be the literal of w present in C_{trg}. Let K' denote the longest clause falsified by $\vec{q}$. Let K'' be a clause formed from $l(w)$ and a subset of literals of C_{trg} such that every clause of $F_{\vec{q}}$ unresolvable with $(C_{trg})_{\vec{q}}$ on w is unresolvable with $(K'')_{\vec{q}}$ too. Let $K_{bct} = K' \vee K''$. Then $(K_{bct})_{\vec{q}} \Rightarrow (C_{trg})_{\vec{q}}$ and K_{bct} is redundant in $\exists X[K_{bct} \wedge (F \setminus \{C_{trg}\})]$.*

Example 1. Let us recall the example of Section 6. Here we have a formula $\exists X[F]$ where $X = \{x_2, x_3\}$, $Y = \{y_1\}$, $F = C_1 \wedge C_2 \wedge C_3 \wedge C_4$, $C_1 = \overline{x}_2 \vee x_3$, $C_2 = y_1 \vee x_2$, $C_3 = y_1 \vee \overline{x}_3$, $C_4 = y_1$. In subspace $y_1 = 1$, the target clause C_1

is *blocked* at x_2 and hence is redundant. (C_1 can be resolved on x_2 only with C_2 that is satisfied by $y_1 = 1$.) This redundancy can be certified by the clause $K_1 = \overline{y_1} \vee \overline{x}_2$ implying C_1 in subspace $y_1 = 1$. The clause K_1 is constructed as $K' \vee K''$ of Proposition 2. Here $K' = \overline{y}_1$ is the clause falsified by the assignment $y_1 = 1$. The clause $K'' = \overline{x}_2$ has the same literal of the blocked variable x_2 as the target clause C_1. (Formula F has no clauses unresolvable with C_1 on x_2. So, K'' needs no more literals.) ■

Remark 3. Let C_{trg} of Proposition 2 be *unit* in subspace $\vec{q}$ (and w be the only unassigned variable of C_{trg}). Then K'' reduces to $l(w)$ and $K_{bct} = K' \vee l(w)$.

8 The Case When The Target Clause Becomes Unit

In this section, we describe what *PrvRed* does when the current target clause C_{trg} becomes unit. (Since *PrvRed* first assigns variables of Y, the unassigned variable of C_{trg} is in X i.e. *quantified.*) In this case, *PrvRed* recursively calls itself to prove redundancy of every clause resolvable with C_{trg}. A concrete example is given in Appendix C.

Figure 4 shows the fragment of *BCP* invoked when the current target C_{trg} becomes unit. Let $x \in X$ denote the only unassigned variable of C_{trg}. Assume for the sake of clarity that C_{trg} contains the positive literal of x. At this point a SAT-solver would derive the assignment $x = 1$ because C_{trg} is falsified under assignment $x = 0$. However, the goal of *PrvRed* is to prove C_{trg} redundant rather than find a satisfying assignment. The fact that C_{trg} is falsified in a subspace says nothing about whether it is redundant there.

```
// $F$ denotes $F_1 \wedge F_2$
//
$BCP(Q, \vec{q}, F, Y, C_{trg})$ {
        ...
10  if ($Unit(C_{trg}, \vec{q})$) {
11     $(K_{bct}, G) := Rcrs(F, C_{trg}, \vec{q})$
12     if ($K_{bct} \neq nil$) return($K_{bct}$)
13     $K_{bct} := GenCert(F, C_{trg}, \vec{q}, G)$
14     return($K_{bct}$) }
        ...
```

Fig. 4: A fragment of *BCP*

So, *BCP* invokes procedure *Rcrs* that recursively calls *PrvRed* for every clause resolvable with C_{trg} on x. The name *Rcrs* abbreviates "recurse". This call can have two outcomes. First, *Rcrs* may return a clause K_{bct} that is falsified by $\vec{q}$. (This is possible only if F is unsatisfiable in subspace $\vec{q}$.) Then *BCP* returns K_{bct} as the reason for backtracking (line 12). Second, *Rcrs* proves the clauses resolvable with C_{trg} on x redundant and returns a set G of certificates. For each clause C resolvable with C_{trg} on x, the set G contains a certificate of redundancy of C in subspace $\vec{q} \cup \{x = 1\}$. At this point, C_{trg} is blocked at x in subspace $\vec{q}$. So, a certificate K_{bct} is built using Proposition 2 (line 13). It is returned by *BCP* as the reason for backtracking.

Remark 4. Every clause C resolvable with C_{trg} on x and proved redundant in subspace $\vec{q} \cup \{x = 1\}$ is *temporarily* removed from the formula F until backtracking. Since C is proved redundant only locally, one has to *return* it to F after backtracking. Nevertheless, C_{trg} remains blocked in subspace $\vec{q}$ and hence redundant there (see Remark 2 of Section 5).

9 Invariant Generation For Bug Detection

In this section, we discuss using PQE for bug detection by invariant generation. An **invariant** P of a sequential circuit N is a formula satisfied by every reachable state of N. So, the states falsifying P are unreachable in N. We will call an invariant **local** if it holds in *some* time frames. To distinguish between local invariants and those holding in every time frame we will call the latter **global**. When we say "invariant" without a qualifier we mean a global invariant.

9.1 Two kinds of bugs

Let $\boldsymbol{N}$ be a sequential circuit. Let $P_1(S)$,...,$P_n(S)$ be invariants that must hold for N where S is the set of state variables. That is, these are *desired* invariants of N. One can view the aggregate invariant $P_1 \wedge \dots \wedge P_n$ as a *specification* $\boldsymbol{Sp}$ for N. We will say that $\vec{s}$ is a **bad state** (respectively a **good state**) if $Sp(\vec{s}) = 0$ (respectively $Sp(\vec{s}) = 1$). As far as reachable states are concerned, N can have **two kinds of bugs**. A *bug of the first kind* occurs when a bad state is reachable in N. A *bug of the second kind* takes place when a good state that is supposed to be reachable is unreachable in N. Informally, a bug of the first kind (respectively the second kind) indicates that the set of reachable states is "larger" (respectively "smaller") than it should be.

To prove that N has no bugs of the first kind, it suffices to show that the aggregate invariant Sp holds for N. Note that this does nothing to identify bugs of the *second* kind. Indeed, let N_{triv} be a circuit looping in an initial state $\vec{s}_{init}$ satisfying Sp. Then Sp holds for N_{triv}. However, N_{triv} has bugs of the second kind (assuming that a correct implementation has to reach states other than $\vec{s}_{init}$). A straightforward way to identify bugs of the second kind is to compute the set of all unreachable states of N. If this set contains a state that is supposed to be reachable, N has a bug of the second kind. Unfortunately, computing such a set can be prohibitively hard.

Note that one cannot *prove* the existence of a bug of the second kind by testing: the unreachability of a state cannot be established by a counterexample. However, testing can point to the possibility of such a bug (see Section 11). An important method for finding bugs of the second kind in a circuit N is to identify its unwanted invariants. We will call Q an **unwanted invariant** if it is falsified by a state $\vec{s}$ that is supposed to be reachable. If Q holds for N, then $\vec{s}$ is unreachable and N has a bug of the second kind. Currently, unwanted invariants are detected via checking a list of expected events [14]. (If an event of this list never occurs, N has an unwanted invariant.) This list is formed *manually.* So, in a sense, unwanted invariants are simply guessed. For instance, one can check if N reaches a state where a state variable $s_i \in S$ changes its initial value. If not, then N has an unwanted invariant, assuming that states with both values of s_i are supposed to be reachable in N. (For the circuit N_{triv} above, this unwanted invariant holds for every state variable.) The problem with guessing unwanted invariants is that, in general, they are as unpredictable as bugs.

In this paper, we consider an approach to finding bugs of the second kind where **invariants are generated** automatically in a systematic way. The necessary condition for an invariant Q to be unwanted is $Sp \not\Rightarrow Q$. (If $Sp \Rightarrow Q$, then Q is a desired invariant of N.) So, the overall idea is to generate invariants of N not implied by Sp and *check if any of them is unwanted.* In some cases, the designer can tell if Q is an unwanted invariant. Otherwise, one needs to find a bug-exposing test as explained in Section 11. In general, an invariant specifies only a subset of unreachable states of N. So, it can be generated much more efficiently than the entire set of unreachable states.

9.2 Invariant generation by PQE

Let us show how one can generate invariants by PQE. First, we consider the generation of a local invariant that holds in k-th time frame. So, a state falsifying such an invariant is unreachable in k transitions. Then we show that a local invariant can be used to generate global invariants. Let formulas I and T specify the initial states and the transition relation of N respectively. Let $\boldsymbol{F_k}$ denote the formula obtained by unfolding N for k time frames. That is $F_k = I(S_0) \wedge T(S_0, S_1) \wedge \cdots \wedge T(S_{k-1}, S_k)$ where S_j denotes the state variables of j-th time frame, $0 \leq j \leq k$. (For the sake of simplicity, in T, we omit the combinational i.e. unlatched variables of N.)

Let $H_k(S_k)$ be a solution to the PQE problem of taking a clause C out of $\exists \mathbb{S}_{k-1}[F_k]$ where $\mathbb{S}_{k-1} = S_0 \cup \cdots \cup S_{k-1}$. That is $\exists \mathbb{S}_{k-1}[F_k] \equiv H_k \wedge \exists \mathbb{S}_{k-1}[F_k \setminus \{C\}]$. Since F_k implies H_k, the latter is a **local invariant** of N holding in k-th time frame. Note that performing full QE on $\exists \mathbb{S}_{k-1}[F_k]$ produces the *strongest* local invariant specifying all states unreachable in k transitions. Computing this invariant can be prohibitively hard. PQE allows to build a collection of *weaker* local invariants H_k each specifying only a subset of states unreachable in k transitions. Computation of such invariants can be dramatically more efficient since PQE can be much easier than QE.

One can use H_k to find global invariants as follows. The fact that H_k is not a global invariant does not mean that *every clause* of H_k is not a global invariant either. On the contrary, the experiments presented in Appendix F showed that even for small k, a large share of clauses of H_k were a global invariant. (To find out if a clause $Q \in H_k$ is a global invariant, one can simply run a model checker to see if Q holds.)

9.3 Using Invariant Generation

One of possible ways to use invariant generation is to take out clauses according to some coverage metric. The intuition here is based on the two observations below. Let Q be an invariant obtained by taking a clause C out of $\exists \mathbb{S}_{k-1}[F_k]$. The first observation is that the states falsifying Q are unreachable due to the *presence* of C. So, if a part of the circuit N is responsible for a bug of the second kind and C is related to this part, taking out C may produce an *unwanted* invariant. This observation is substantiated in the next section. The second observation is that by taking out different clauses one generates different invariants "covering" different parts of the circuit N. An example of a coverage metric is presented in the next section. There we take out the clauses containing an unquantified variable of $\exists \mathbb{S}_{k-1}[F_k]$ (i.e. a state variable of the k-th time frame). One can view such a choice of clauses as a way to cover the design in terms of *latches*.

10 An Experiment With FIFO Buffers

In this section, we describe an experiment with FIFO buffers. Our objective here is twofold. First, we explain how bug detection by invariant generation works on a practical example. Second, we want to show that even the current version of $START$ whose performance can be dramatically improved can address an important practical problem. (A mature PQE algorithm can be applied to a long list of problems including those listed in Section 4.) In Appendix F, we apply $START$ to invariant generation for HWMCC-13 benchmarks. We also use these benchmarks to compare PQE with QE and $START$ with *DS-PQE*, our previous PQE-solver [1]. In this section, such a comparison is done on FIFO buffers. A Linux binary of $START$ and a sample of formulas used in the experiments can be downloaded from [12]. In all experiments, we used a computer with Intel Core i5-8265U CPU of 1.6 GHz.

10.1 Buffer description

```
     ...
if (write == 1 && currSize < n)
* if (dataIn != Val)
    begin
    Data[wrPnt] = dataIn;
    wrPnt = wrPnt + 1;
    end
     ...
```

Fig. 5: A buggy fragment of Verilog code describing *Fifo*

In this section, we give an example of bug detection by invariant generation for a FIFO buffer called *Fifo*. Let n be the number of elements of *Fifo* and *Data* denote the data buffer of *Fifo*. Let each $\mathit{Data}[i], i = 1, \ldots, n$ have p bits and be an integer where $0 \leq \mathit{Data}[i] < 2^p$. A fragment of the Verilog code describing *Fifo* is shown in Fig 5. This fragment has a buggy line marked with an asterisk. In the correct version without the marked line, a new element *dataIn* is added to *Data* if the *write* flag is on and *Fifo* holds less than n elements. Since *Data* can have any combination of numbers, all *Data* states are supposed to be reachable.

However, due to the bug, the number *Val* cannot appear in *Data*. (Here *Val* is some constant $0 < \mathit{Val} < 2^p$. We assume that the buffer elements are initialized to 0.) So, *Fifo* has a *bug of the second kind* since it cannot reach states where an element of *Data* equals *Val*. This bug is hard to detect by random testing because it is exposed only if one tries to add *Val* to *Fifo*. Similarly, it is virtually impossible to guess an unwanted invariant of *Fifo* exposing this bug unless one knows exactly what this bug is.

10.2 Bug detection by invariant generation

Let N be a circuit implementing *Fifo*. Let S be the set of state variables of N and $\boldsymbol{S_{data}} \subset \boldsymbol{S}$ be the subset corresponding to the data buffer *Data*. We used $START$ to generate invariants of N as described in the previous section. Note that an invariant Q depending only on S_{data} is an unwanted one. If Q holds for N, some states of *Data* are unreachable. Then *Fifo* has a *bug of the second kind* since every state of *Data* is supposed to be reachable. To generate invariants, we used the formula $F_k = I(S_0) \wedge T(S_0, S_1) \wedge \cdots \wedge T(S_{k-1}, S_k)$ introduced in Subsection 9.2. Here I and T describe the initial states and the transition relation of N respectively and S_j is the set of state variables in j-th time frame. First, we used $START$ to generate local invariants H_k. Namely, H_k was obtained by taking a clause C out of $\exists \mathbb{S}_{k-1}[F_k]$ where $\mathbb{S}_{k-1} = S_0 \cup \cdots \cup S_{k-1}$. That is, $\exists \mathbb{S}_{k-1}[F_k] \equiv H_k \wedge \exists \mathbb{S}_{k-1}[F_k \setminus \{C\}]$. We picked clauses to take out as described in Subsection 9.3. Namely, we took out only clauses containing an unquantified variable (i.e. a state variable of the k-th time frame). The time limit for solving the PQE problem of taking out a clause was set to 10 sec.

For each clause Q of every local invariant H_k generated by PQE, we checked if Q was a global invariant. Namely, we used a publicly available version of *IC3* [15,16] to verify if the invariant Q held. If so, and Q depended only on variables of S_{data}, N had an *unwanted invariant.* Then we stopped invariant generation. The results of the experiment are given in Table 1. (In the experiment, we considered buffers with 32-bit elements.) Let us use the first line of Table 1 to explain its structure. The first two columns show the size of *Fifo* implemented by N and the number of latches in N (8 and 300). The third column gives the number k of time frames (i.e. 5). The value 13 shown in the fourth column is the number of clauses taken out of $\exists \mathbb{S}_{k-1}[F_k]$ before an unwanted invariant was generated. That is, 13 was the number of PQE problems for $START$ to solve.

Table 1: FIFO buffer with n elements of 32 bits. Time limit is 10 sec. per PQE problem

buff. size (n)	latches	time frames	clauses taken out	local single clause invariants			tot. run time (s.)
				gen. invar.	global?		
					no	yes	
8	300	5	13	10	8	2	25
8	300	10	11	4	1	3	54
16	560	5	26	18	16	2	43
16	560	10	17	2	0	2	78

Let C be a clause taken out of the scope of quantifiers by $START$. Every free clause Q generated when taking out C was stored as a local *single-clause invariant.* The fifth column shows that when solving the 13 PQE problems above, $START$ generated 10 free clauses forming 10 local single-clause invariants. These invariants held in k-th time frame (where $k\!=\!5$). The next two columns show how many invariants out of 10 *IC3* proved false or true *globally* (8 and 2). The last column gives the total run time (25 sec).

For all four instances of *Fifo* listed in Table 1, the invariants generated by $START$ had one asserting that *Fifo* cannot reach a state where an element of *Data* equals *Val*. This invariant was produced when taking out a clause of F_k related to the buggy line of Fig. 5 (confirming the intuition of Subsection 9.3.)

When picking a clause to take out, i.e. a clause containing a state variable of k-th time frame, one could make a good choice by pure luck. To address this issue, we picked clauses to take out *randomly* and performed 10 different runs of invariant generation. For each line of Table 1, the columns four to eight actually describe the average value of 10 runs.

10.3 Comparing PQE and QE

To contrast PQE and QE, we used a high-quality tool *CADET* [17,18] to perform QE on formulas $\exists \mathbb{S}_{k-1}[F_k]$. That is, instead of taking a clause out of $\exists \mathbb{S}_{k-1}[F_k]$ by PQE, we applied *CADET* to perform full QE on this formula. As mentioned in Subsection 9.2, performing QE on $\exists \mathbb{S}_{k-1}[F_k]$ produces the strongest local invariant specifying all states unreachable in k transitions. *CADET* failed to finish QE on $\exists \mathbb{S}_{k-1}[F_k]$ with the time limit of 600 sec. On the other hand, *START* finished 63% of the PQE problems of taking a clause out of $\exists \mathbb{S}_{k-1}[F_k]$ in the time limit (i.e. under 10 sec). This shows that PQE can be dramatically more efficient than QE if only a small part of the formula gets unquantified.

10.4 *START* versus *DS-PQE*

We repeated the experiment above using *DS-PQE* instead of *START*. *DS-PQE* is our previous PQE-solver [1] based on the machinery of D-sequents [19,20]. *DS-PQE* solved only 2% of the PQE problems in the time limit of 10 sec. (as opposed to 63% by *START*) and failed to generate an unwanted invariant.

11 Identifying Unwanted Invariants

Sometimes it is easy to see that an invariant Q is unwanted (e.g. an invariant of *Fifo* depending only on variables of S_{data} is obviously unwanted). However, in general, to show that Q is unwanted, one needs to find a **bug-exposing** test (or **be-test** for short.) Let $\vec{t}$ denote a test $(\vec{s}_0, \vec{x}_0, \dots, \vec{x}_{k-1})$ for a circuit N. Here $\vec{s}_0$ is an initial state of N and $\vec{x}_i$, $0 \leq i < k$ is a full assignment to the combinational input variables of N in i-th time frame. (Recall that so far, for the sake of simplicity, we omitted combinational variables in the description of N.)

Let $(\vec{s}_0, \vec{s}_1, \dots, \vec{s}_k)$ be the trace produced by the test $\vec{t}$ above (i.e. N moves from state $\vec{s}_i$ to $\vec{s}_{i+1}$ under input $\vec{x}_i$). We will say that $\vec{t}$ is a **be-test** for an invariant Q if it is a *counterexample* for Q in a *correct* version N' of N. That is $\vec{t}$ produces a trace $(\vec{s}_0, \vec{s'}_1, \dots, \vec{s'}_k)$ in N' where $\vec{s'}_k$ *falsifies* Q. Consider, for instance, the invariant Q stating that *Fifo* cannot have the number *Val* in j-th element of its data buffer. Let $\vec{t} = (\vec{s}_0, \vec{x}_0, \dots, \vec{x}_{k-1})$ be a test such that when applied to a correct design, $\vec{t}$ would make *Val* appear in the j-th element of the data buffer. Then $\vec{t}$ is a be-test for Q.

Finding a be-test is based on the following idea. Let an invariant clause Q be extracted from a formula H_k obtained by taking a clause C out of $\exists \mathbb{S}_{k-1}[F_k]$ as described above. As we mentioned in Remark 1, $\exists \mathbb{S}_{k-1}[F_k] \equiv H_k \wedge \exists \mathbb{S}_{k-1}[F_k \setminus \{C\}]$ holds even if the clauses implied by $F_k \setminus \{C\}$ are removed from H_k. So, we will assume that $F_k \setminus \{C\} \not\Rightarrow Q$. Then there is an assignment $\vec{p}$ satisfying $(F_k \setminus \{C\}) \wedge \overline{Q}$. One can view $\vec{p}$ as an execution trace of N when C is removed from F_k.

Let $\vec{t^*} = (\overrightarrow{s^*}_0, \overrightarrow{x^*}_0, \dots, \overrightarrow{x^*}_{k-1})$ be the test where the variables are assigned as in $\vec{p}$. One can make two claims about $\vec{t^*}$. First, if Q is an unwanted invariant, $\vec{t^*}$ can be *very close to a be-test*. Second, if Q is a *desired* invariant, $\vec{t^*}$ is a *high-quality test* for N that can be used e.g. in regression testing. The first claim is due to $\vec{t^*}$ being extracted from $\vec{p}$ falsifying Q and satisfying all clauses of F_k but C. The second claim is due to $\vec{t^*}$ being able to detect modifications of N breaking Q. One can try to produce a be-test from $\vec{t^*}$ either "manually" or automatically generating small variations of $\vec{t^*}$.

Consider, for example, the unwanted invariant Q stating that the number *Val* cannot appear in j-th element of the data buffer of *Fifo*. For every example of Table 1, we built the test $\vec{t^*} = (\overrightarrow{s^*}_0, \overrightarrow{x^*}_0, \dots, \overrightarrow{x^*}_{k-1})$ extracted from $\vec{p}$ satisfying $(F_k \setminus \{C\}) \wedge \overline{Q}$. In every case, $\vec{t^*}$ turned out to be different from a be-test only *in one bit*.

12 Some Background

In this section, we discuss some research relevant to PQE and invariant generation. Information on BDD and SAT based QE can be found in [21,22] and [23,24,25,26,27,28,29,30,17] respectively. Making clauses of a formula redundant by adding resolvents is routinely used in pre-processing [31,32] and in-processing [33] phases of QBF/SAT-solving. Identification and removal of blocked clauses is also an important part of formula simplification [34]. The difference of our approach from these techniques is twofold. First, our approach employs redundancy based *reasoning* rather than formula optimization. So, for instance, to make a target clause redundant, *START* can add a lot of new clauses making the formula *larger*. Second, these techniques try to identify *non-trivial* conditions under which a clause C is redundant in the *entire space*. In our approach, one *branches* to reach a subspace where proving C redundant is *trivial*. Proving redundancy of C in the entire space is achieved by merging the results of different branches.

The predecessor of the approach based on certificate clauses is the machinery of dependency sequents (D-sequents) [19,20]. A D-sequent is a record stating redundancy of a clause in a quantified formula. A flaw of this machinery is that to reuse a learned D-sequent, one has to keep a lot of contextual information [35], which makes D-sequent reusing expensive. On the other hand, the reuse of certificate clauses does not require to store any contextual information.

To the best of our knowledge, the existing procedures generate only particular classes of invariants. For instance, they generate invariants relating internal points of circuits to check for equivalence [36] or loop invariants [37]. Another example of special invariants are clauses generated by *IC3* to make an invariant

P inductive [15]. The problem here is that the closer P to an inductive invariant, the fewer invariant clauses *IC3* generates to make P inductive. For instance, for the circuit N_{triv} mentioned in Subsection 9.1 that loops in an initial state, every true desired invariant P_i is *already inductive*. Hence, *IC3* will not generate any new invariant clauses and will not produce an unwanted invariant even though N_{triv} is obviously buggy. In Appendix F.3, we experimentally compare invariants generated by *IC3* and *START*.

13 Conclusions

We consider *partial* quantifier elimination (PQE) on propositional CNF formulas with existential quantifiers. PQE allows to unquantify a *part* of the formula. We present a PQE algorithm called *START* employing redundancy based reasoning via the machinery of certificate clauses. To prove a target clause C redundant, *START* derives a clause implying C, thus "certifying" its redundancy. The version of *START* we describe here can still be drastically improved. We show that PQE can be used to generate invariants of a sequential circuit. The goal of invariant generation is to find an *unwanted* invariant of this circuit indicating that the latter is buggy. Bugs causing unwanted invariants can be easily overlooked by the existing methods. We applied *START* to identify a bug in a FIFO buffer by generating an unwanted invariant of this buffer. We also showed that even the current version of *START* is good enough to generate invariants for HWMCC-13 benchmarks. Our experiments suggest that *START* can be used for detecting hard-to-find bugs in real-life designs.

References

1. E. Goldberg and P. Manolios, "Partial quantifier elimination," in *Proc. of HVC-14*. Springer-Verlag, 2014, pp. 148–164.
2. Introduction to partial quantifier elimination, https://eigold.tripod.com/pqe_page.pdf.
3. W. Craig, "Three uses of the herbrand-gentzen theorem in relating model theory and proof theory," *The Journal of Symbolic Logic*, vol. 22, no. 3, pp. 269–285, 1957.
4. K. McMillan, "Interpolation and sat-based model checking," in *CAV-03*. Springer, 2003, pp. 1–13.
5. E. Goldberg, "Property checking by logic relaxation," Tech. Rep. arXiv:1601.02742 [cs.LO], 2016.
6. E. Goldberg and P. Manolios, "Software for quantifier elimination in propositional logic," in *ICMS-2014,Seoul, South Korea, August 5-9*, 2014, pp. 291–294.
7. E. Goldberg, "Equivalence checking by logic relaxation," in *FMCAD-16*, 2016, pp. 49–56.
8. ——, "Property checking without inductive invariant generation," Tech. Rep. arXiv:1602.05829 [cs.LO], 2016.
9. G. Tseitin, "On the complexity of derivation in the propositional calculus," *Zapiski nauchnykh seminarov LOMI*, vol. 8, pp. 234–259, 1968, english translation of this volume: Consultants Bureau, N.Y., 1970, pp. 115–125.

10. A. Biere, A. Cimatti, E. Clarke, M. Fujita, and Y. Zhu, "Symbolic model checking using sat procedures instead of bdds," in *DAC*, 1999, pp. 317–320.
11. O. Kullmann, "New methods for 3-sat decision and worst-case analysis," *Theor. Comput. Sci.*, vol. 223, no. 1-2, pp. 1–72, 1999.
12. "A linux binary of start and some instances of pqe problems," http://eigold.tripod.com/software/start.tar.gz.
13. J. Marques-Silva and K. Sakallah, "Grasp – a new search algorithm for satisfiability," in *ICCAD-96*, 1996, pp. 220–227.
14. B. Cohen, S. Venkataramanan, A. Kumari, and L. Piper, *SystemVerilog Assertions Handbook: ... For Dynamic and Formal Verification*, 4th ed. North Charleston, SC, USA: CreateSpace Independent Publishing Platform, 2015.
15. A. R. Bradley, "Sat-based model checking without unrolling," in *VMCAI*, 2011, pp. 70–87.
16. An implementation of IC3 by A. Bradley, https://github.com/arbrad/IC3ref.
17. M. Rabe, "Incremental determinization for quantifier elimination and functional synthesis," in *CAV*, 2019.
18. CADET, https://github.com/MarkusRabe/cadet.
19. E. Goldberg and P. Manolios, "Quantifier elimination by dependency sequents," in *FMCAD-12*, 2012, pp. 34–44.
20. ——, "Quantifier elimination via clause redundancy," in *FMCAD-13*, 2013, pp. 85–92.
21. R. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Transactions on Computers*, vol. C-35, no. 8, pp. 677–691, August 1986.
22. P. Chauhan, E. Clarke, S. Jha, J. Kukula, H. Veith, and D. Wang, "Using combinatorial optimization methods for quantification scheduling," ser. CHARME-01, 2001, pp. 293–309.
23. K. McMillan, "Applying sat methods in unbounded symbolic model checking," in *Proc. of CAV-02*. Springer-Verlag, 2002, pp. 250–264.
24. H. Jin and F.Somenzi, "Prime clauses for fast enumeration of satisfying assignments to boolean circuits," ser. DAC-05, 2005, pp. 750–753.
25. M. Ganai, A.Gupta, and P.Ashar, "Efficient sat-based unbounded symbolic model checking using circuit cofactoring," ser. ICCAD-04, 2004, pp. 510–517.
26. J. Jiang, "Quantifier elimination via functional composition," in *Proceedings of the 21st International Conference on Computer Aided Verification*, ser. CAV-09, 2009, pp. 383–397.
27. J. Brauer, A. King, and J. Kriener, "Existential quantification as incremental sat," ser. CAV-11, 2011, pp. 191–207.
28. W. Klieber, M. Janota, J.Marques-Silva, and E. Clarke, "Solving qbf with free variables," in *CP*, 2013, pp. 415–431.
29. N. Bjorner, M. Janota, and W. Klieber, "On conflicts and strategies in qbf," in *LPAR*, 2015.
30. N. Bjorner and M. Janota, "Playing with quantified satisfaction," in *LPAR*, 2015.
31. N. Eén and A. Biere, "Effective preprocessing in sat through variable and clause elimination," in *SAT*, 2005, pp. 61–75.
32. A. Biere, F. Lonsing, and M. Seidl, "Blocked clause elimination for qbf," ser. CADE-11, 2011, pp. 101–115.
33. M. Järvisalo, M. Heule, and A. Biere, "Inprocessing rules," ser. IJCAR-12, 2012, pp. 355–370.
34. M. Järvisalo, A. Biere, and M. Heule, "Blocked clause elimination," in *TACAS*, 2010, pp. 129–144.

35. E. Goldberg, “Quantifier elimination with structural learning,” Tech. Rep. arXiv: 1810.00160 [cs.LO], 2018.
36. J.Baumgartner, H. Mony, M. Case, J. Sawada, and K. Yorav, “Scalable conditional equivalence checking: An automated invariant-generation based approach,” in *2009 Formal Methods in Computer-Aided Design*, 2009, pp. 120–127.
37. I. Dillig, T. Dillig, B. Li, and K. McMillan, “Inductive invariant generation via abductive inference,” vol. 48, 10 2013, pp. 443–456.
38. HardWare Model Checking Competition 2013 (HWMCC-13), http://fmv.jku.at/hwmcc13/.
39. R. E. Bryant, “Symbolic Boolean manipulation with ordered binary decision diagrams,” *ACM Computing Surveys*, 1992.

Appendix

A Proofs

Lemma 1 is used in the proof of Proposition 1.

Lemma 1. *Given a formula $\exists X[F(X)]$, let C be a clause blocked in $\exists X[F]$ at w. Then $\exists X[F] \equiv \exists X[F \setminus \{C\}]$ i.e. C is redundant in $\exists X[F]$.*

Proof. Let us prove that F and $F \setminus \{C\}$ are equisatisfiable (and so $\exists X[F] \equiv \exists X[F \setminus \{C\}]$). The satisfiability of F obviously implies that of $F \setminus \{C\}$. Let us show that the converse is true as well. Let $\vec{x}$ be a full assignment to X satisfying $F \setminus \{C\}$. If $\vec{x}$ satisfies C, it satisfies F and our proof is over. Now assume that $\vec{x}$ falsifies C and hence falsifies F. Let $\vec{x}_{fl}$ be the assignment obtained from $\vec{x}$ by flipping the value of w. (So $\vec{x}_{fl}$ satisfies C.) Let G be the set of clauses of F resolvable with C on w. Let $w = b$ satisfy C where $b \in \{0, 1\}$. (So, w is assigned b in $\vec{x}_{fl}$, because $\vec{x}$ falsifies C.)

First, let us show that $\vec{x}_{fl}$ satisfies $F \setminus G$. Assume the contrary i.e. $\vec{x}_{fl}$ falsifies a clause D of $F \setminus G$. (Note that D is different from C because the latter is satisfied by $\vec{x}_{fl}$.) 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 $F \setminus \{C\}$. 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.

Since $\vec{x}_{fl}$ satisfies $F \setminus G$, then $(F \setminus G)_{w=b}$ is satisfiable. By definition of a blocked clause (see Definition 10), G is redundant in $\exists X[F]$ in subspace $w = b$. So formula $F_{w=b}$ is satisfiable. Hence F is satisfiable too.

Proposition 1. *Given a formula $\exists X[F(X,Y)]$, let C be a clause blocked in $\exists X[F]$ at $w \in X$. Then C is redundant in $\exists X[F]$ i.e. $\exists X[F] \equiv \exists X[F \setminus \{C\}]$. So, C is es-implied by $F \setminus \{C\}$ in $\exists X[F]$.*

Proof. One needs to show that for every full assignment $\vec{y}$ to Y, formulas $F_{\vec{y}}$ and $(F \setminus \{C\})_{\vec{y}}$ are equisatisfiable. If $\vec{y}$ satisfies C, it is trivially true. Assume that $\vec{y}$ does not satisfy C. From Definition 10 it follows that if C is blocked in $\exists X[F]$ at a variable w, then $C_{\vec{y}}$ is blocked in $(\exists F[X])_{\vec{y}}$ at w. Then from Lemma 1 if follows that $C_{\vec{y}}$ is redundant in $(\exists F[X])_{\vec{y}}$

Proposition 2. *Given a formula $\exists X[F(X,Y)]$, let $C_{trg} \in F$. Let $\vec{q}$ be an assignment to $X \cup Y$ that does not satisfy C_{trg}. Let C_{trg} be blocked in $\exists X[F]$ at $w \in X$ in subspace $\vec{q}$ where $w \not\in \mathit{Vars}(\vec{q})$. Let $l(w)$ be the literal of w present in C_{trg}. Let K' denote the longest clause falsified by $\vec{q}$. Let K'' be a clause formed from $l(w)$ and a subset of literals of C_{trg} such that every clause of $F_{\vec{q}}$ unresolvable with $(C_{trg})_{\vec{q}}$ on w is unresolvable with $(K'')_{\vec{q}}$ too. Let $K_{bct} = K' \vee K''$. Then $(K_{bct})_{\vec{q}} \Rightarrow (C_{trg})_{\vec{q}}$ and K_{bct} is redundant in $\exists X[K_{bct} \wedge (F \setminus \{C_{trg}\})]$.*

Proof. The fact that $(K_{bct})_{\vec{q}} \Rightarrow (C_{trg})_{\vec{q}}$ trivially follows from the definition of K_{bct}. The latter equals $K' \vee K''$ where K' is falsified by $\vec{q}$ and K'' consists only of (some) literals of C_{trg}. Now we prove that the clause K_{bct} is redundant in $\exists X[K_{bct} \wedge (F \setminus \{C_{trg}\})]$. Let H denote $F \setminus \{C_{trg}\}$. One needs to show that for every full assignment $\vec{y}$ to Y, $(K_{bct} \wedge H)_{\vec{y}}$ and $H_{\vec{y}}$ are equisatisfiable. If $\vec{y}$ satisfies K_{bct}, it is trivially true. Let $\vec{y}$ not satisfy K_{bct}. (This means that the variables of $\mathit{Vars}(\vec{y}) \cap \mathit{Vars}(\vec{q})$, if any, are assigned the same value in $\vec{y}$ and $\vec{q}$.) The satisfiability of $(K_{bct} \wedge H)_{\vec{y}}$ obviously implies that of $H_{\vec{y}}$. Below, we show *in three steps* that the converse is true as well. First, we introduce an assignment $\vec{p}_{fl}$ such that $\vec{y} \subseteq \vec{p}_{fl}$ and $\vec{q} \subseteq \vec{p}_{fl}$. (Here 'fl' stands for 'flipped'.) Second, we prove that $\vec{p}_{fl}$ satisfies $F_{\vec{q}} \setminus G_{\vec{q}}$ where G is the set of clauses resolvable with C_{trg}. Third, we show that the satisfiability of $F_{\vec{q}} \setminus G_{\vec{q}}$ and the fact that C_{trg} is blocked imply that $(K_{bct} \wedge H)_{\vec{y}}$ is satisfiable.

Step 1. Let $\vec{p}$ denote a full assignment to $X \cup Y$ such that $\vec{y} \subseteq \vec{p}$ and $\vec{p}$ satisfies $H_{\vec{y}}$. If $\vec{p}$ satisfies $(K_{bct} \wedge H)_{\vec{y}}$, our proof is over. Assume that $\vec{p}$ falsifies $(K_{bct} \wedge H)_{\vec{y}}$. Then $\vec{p}$ falsifies K_{bct}. This means that $\vec{q} \subseteq \vec{p}$. Let $w = b$ denote the assignment to w in $\vec{p}$. Denote by $\vec{p}_{fl}$ the assignment obtained from $\vec{p}$ by flipping the value of w from b to $\overline{b}$. Denote by $\vec{q}_{ext}$ the extension $\vec{y} \cup \vec{q} \cup \{w = \overline{b}\}$ of the assignment $\vec{q}$. Note that $\vec{q}_{ext} \subseteq \vec{p}_{fl}$. Besides, due to the assignment $w = \overline{b}$, both $\vec{q}_{ext}$ and $\vec{p}_{fl}$ satisfy K_{bct} and C_{trg}.

Step 2. Let G denote the set of clauses of F resolvable with C_{trg} on w. Then $G_{\vec{q}}$ is the set of clauses of $F_{\vec{q}}$ resolvable with $(C_{trg})_{\vec{q}}$ on w. Let us show that $\vec{p}_{fl}$ satisfies $F_{\vec{q}} \setminus G_{\vec{q}}$. Assume the contrary i.e. there is a clause $D \in F_{\vec{q}} \setminus G_{\vec{q}}$ falsified by $\vec{p}_{fl}$. First, assume that D does not contain w. Then D is falsified by $\vec{p}$ as well. So, $\vec{p}$ falsifies $F_{\vec{q}}$ and hence $H_{\vec{q}}$ (because $(C_{trg})_{\vec{q}}$ is satisfied by $\vec{p}_{fl}$ and so is different from D). Thus, we have a contradiction. Now, assume that D contains the literal $\overline{l(w)}$. Then it is resolvable with clause $(K_{bct})_{\vec{q}}$. This means that D is resolvable with $(C_{trg})_{\vec{q}}$ too. (By our assumption, every clause of $F_{\vec{q}}$ unresolvable with $(C_{trg})_{\vec{q}}$ is unresolvable with $(K_{bct})_{\vec{q}}$ too.) Then D cannot be in $F_{\vec{q}} \setminus G_{\vec{q}}$ and we have a contradiction.

Step 3. Since $\vec{p}_{fl}$ satisfies $F_{\vec{q}} \setminus G_{\vec{q}}$, the formula $F_{\vec{q}_{ext}} \setminus G_{\vec{q}_{ext}}$ is satisfiable. The same applies to $(F \setminus G)_{\vec{q}_{ext}}$. Since C_{trg} is blocked in $\exists X[F]$ at w in subspace $\vec{q}$, it is also blocked in $\exists X[F]$ in subspace $\vec{y} \cup \vec{q}$. Then $(F \setminus G)_{\vec{q}_{ext}}$ es-implies $G_{\vec{q}_{ext}}$ (see Definition 10) and $F_{\vec{q}_{ext}}$ is satisfiable too. Since K_{bct} is satisfied by

$\vec{q}_{ext}$, then $(K_{bct} \wedge F)_{\vec{q}_{ext}}$ is satisfiable. Hence $(K_{bct} \wedge F)_{\vec{y}}$ is satisfiable and so is $(K_{bct} \wedge H)_{\vec{y}}$

B Backtracking By *START*

When a SAT-solver encounters a conflict, it generates a conflict clause and backtracks to the smallest decision level where this clause is unit. So, an assignment can be derived from this clause. In contrast to a SAT-solver, the goal of a PQE-solver is to prove a target clause C_{trg} redundant rather than find a satisfying assignment. So, *START* backtracks slightly differently from a SAT-solver. After *START* derives a certificate K proving C_{trg} in the current subspace, it backtracks to the smallest decision level at which the *conditional* of the derived certificate K (rather than K itself) is unit.

Definition 11. *Let K be a certificate stating the redundancy of clause C_{trg} in a subspace. The clause consisting of the literals of K not shared with C_{trg} is called the* ***conditional of K****.*

```
1   Find $F_1^*(Y)$ such that
2   $\exists X[F_1 \wedge F_2] \equiv F_1^* \wedge \exists X[F_2]$
3   $Y = \{y_1\}, X = \{x_2, x_3, x_4, \dots\}$
4   $F_1 = \{C_1\}$, $C_1 = x_2 \vee x_4$
5   $F_2 = \{C_2, C_3, \dots\}, C_2 = y_1 \vee x_3,$
6       $C_3 = \overline{x}_3 \vee x_4, \dots$
7   pick. $C_1 \in F_1$ to prove red.
8   — call $PrvRed$—
9   decision: $y_1 = 0$ at level 1
10  $BCP$:$(C_2 : x_3 = 1)$
11        $C_3 = x_4$ in curr. subsp.
12  LEAF: $C_3$ impl. $C_1$ at level 1
13  $K_1 = y_1 \vee x_4$ is the final cert.
14  $K_1 = Res(C_3, C_2, x_3)$
15  backtracking to level 0
16  $BCP$: $(K_1 : y_1 = 1)$
17  ....
```

Fig. 6: Backtracking by *START*

If the conditional of K is empty, K implies C_{trg} in the entire space. Otherwise, K implies C_{trg} only in subspaces $\vec{q}$ where the conditional of K is falsified by $\vec{q}$. 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.

Example 2. Consider the example of Fig. 6 showing the operation of *START*. This figure gives only the relevant part of formula F_2 and the relevant fragment of the execution trace. *PrvRed* begins proving the target clause $C_1 = x_2 \vee x_4$ redundant by the decision assignment $y_1 = 0$. Then it calls *BCP* that derives $x_3 = 1$ from the clause C_2. At this point, C_3 becomes the unit clause x_4 implying C_1. So, *BCP* returns C_3 as the reason for backtracking (i.e. as the clause K_{bct} in line 6 of Fig. 3). Then the *Lrn* procedure generates the final certificate $K_1 = y_1 \vee x_4$ by resolving C_3 and C_2 to drop the non-decision variable x_3 assigned at level 1 (line 14).

The *conditional* of K_1 is the unit clause y_1 because the literal x_4 is shared by K_1 and the target clause C_1. *PrvRed* backtracks to level 0, the smallest level where the conditional of K_1 is unit. (Note that K_1 itself is not unit at level 0). Then *PrvRed* runs *BCP* and derives the assignment $y_1 = 1$ from K_1 even though K_1 is not unit at level 0. This derivation is possible because K_1 certifies that the redundancy of C_1 in subspace $y_1 = 0$ is *already proved* ■

As we mentioned above, in the general case, after deriving a certificate K, *PrvRed* backtracks to the smallest decision level where the conditional of K is unit. The assignment derived from K is added to the assignment queue Q (lines 10 and 12 of Fig. 3). If K shares no literals with C_{trg}, the *PrvRed* procedure backtracks as a regular SAT-solver, i.e. to the smallest decision level where K is unit.

C Operation Of *START* When C_{trg} Becomes Unit

```
1  Find $F_1^*(Y)$ such that
2  $\exists X[F_1 \wedge F_2] \equiv F_1^* \wedge \exists X[F_2]$
3  $Y = \{y_1\}, X = \{x_2, x_3\}$
4  $F_1 = \{C_1\}$, $C_1 = y_1 \vee x_2$
5  $F_2 = \{C_2, C_3\}$,$C_2 = \overline{x}_2 \vee x_3$,
6  $C_3 = \overline{y}_1 \vee \overline{x}_3$
7  putting $C_1$ to target level $A$
8  PrvRed: proving $C_1$ redund.
9  dec.: $y_1 = 0$ at dec. level 1
10    BCP: $C_1$ is unit at dec. level 1
11    creating target level $B$
12    of clauses res. with $C_1$ on $x_2$
13    making impl. assign. $x_2 = 1$
14    picking $C_2$ as a new targ.
15    PrvRed: prov. $C_2$ redund.
16    in subsp. $y_1 = 0, x_2 = 1$
17    LEAF: $C_2$ is blocked at $x_3$
18    Der. cert. $K' = y_1 \vee x_3$
19    $C_2$ is red. in subsp. above
20    $C_2$ is temporarily removed
21 eliminating targ. level B
22 undoing $x_2 = 1$
23 $C_2$ is restored in the formula
24 LEAF: $C_1$ is blocked at $x_2$
25 der. cert. $K'' = y_1 \vee x_2$
        ...
```

Fig. 7: C_{trg} becomes unit

In Section 8, we described how *START* operates when the current target clause C_{trg} becomes unit. In this appendix, we give a concrete example. Consider solving the PQE problem shown in Fig. 7 by lines 1-6. First, C_1 is picked as a clause to prove. We will refer to it as the **primary** target assuming that it makes up target level A. After decision assignment $y_1 = 0$, the clause C_1 turns into unit clause x_2 (lines 9-10). **Denote** the current assignment (i.e. $y_1 = 0$) as $\vec{q}$. At this point, a SAT-solver would simply derive the assignment $x_2 = 1$. However, the goal of *PrvRed* is not to check if $F_1 \wedge F_2$ is satisfiable but to prove C_1 redundant. The fact that C_1 is falsified in subspace $\vec{q} \cup \{x_2 = 0\}$ does not say anything about whether C_1 is *redundant* there.

So, *PrvRed* creates a new target level (referred to as level B). It consists of the clauses resolvable with C_1 on x_2. Suppose all clauses of this level are redundant in subspace $\vec{q} \cup \{x_2 = 1\}$. Then according to Definition 10, C_1 is blocked (and hence redundant) in $\exists X[F_1 \wedge F_2]$ in subspace $\vec{q}$. In our case, level B consists only of C_2. So, *PrvRed* recursively calls itself to prove redundancy of C_2 in subspace $\vec{q} \cup \{x_2 = 1\}$ (lines 15-20). Note that C_2 is blocked at x_3 in this subspace since C_3 (the clause resolvable with C_2 on x_3) is satisfied by $y_1 = 0$. Then using Proposition 2, *PrvRed* derives the certificate $K' = y_1 \vee x_3$ asserting the redundancy of C_2. The latter is temporarily removed from the formula as redundant (see Remark 4). At this point, the second activation of *PrvRed* terminates.

Then the first activation of *PrvRed* undoes target level B and assignment $x_2 = 1$. The clause C_2 is restored in the formula (lines 21-23). Now, the primary target C_1 is blocked at x_2, since C_2 is proved redundant in subspace $\vec{q} \cup \{x_2 = 1\}$. Using Proposition 2, *PrvRed* derives the certificate $K'' = y_1 \vee x_2$ proving redundancy of C_1 in the entire space.

D Certificate Generation When A Conflict Occurs

```
1  Find $F_1^*(Y)$ such that
2  $\exists X[F_1 \wedge F_2] \equiv F_1^* \wedge \exists X[F_2]$
3  $Y=\{y_1\}, X=\{x_2,x_3\}$
4  $F_1 = \{C_1\}$, $C_1 = \overline{x}_2 \vee x_3$
5  $F_2=\{C_2,C_3\}$,$C_2=\overline{y}_1 \vee x_2$,
6      $C_3=\overline{y}_1 \vee \overline{x}_3$
7  pick. $C_1 \in F_1$ to prove red.
8  — call PrvRed—
9  decision: $y_1 = 0$ at level 1
10 LEAF: $C_1$ is blocked at $x_2$
11 (since $C_2$ is sat. by $y_1 = 0$)
12 $K_1=y_1 \vee \overline{x}_2$ is a witness cert.
13 $K_1$ is not added to $F_1 \wedge F_2$
14 backtracking to level 0
15 BCP: $(K_1 : y_1 = 1)$
16         $(C_2 : x_2 = 1)(C_3 : x_3 = 0)$
17 LEAF: conflict at level 0
18 $C_1=\overline{x}_2 \vee x_3$ is falsified
19 $\hat{K} = \overline{y}_1$ is a new clause
20     falsif. in curr. subspace
21     $R_1 = \mathit{Res}(C_1,C_2,x_2)$,
22     $\hat{K} = \mathit{Res}(R_1,C_3,x_3)$
23 $\hat{K}$ is added to $F_1 \wedge F_2$
24 $K_2=\overline{x}_2$ is a witness cert.
25     $K_2=\mathit{Res}(K_1,\hat{K},y_1)$
26 $K_2$ is not added to $F_1 \wedge F_2$
       . . .
```

Fig. 8: Adding a special clause after a conflict

In this appendix, we discuss in more detail the generation of a certificate by the *Lrn* procedure when a conflict occurs. As before, we denote $F_1 \wedge F_2$ by F. Let C_{trg} be the current target clause. Let K_{bct} be the clause of F falsified in this conflict. (Here, we use the notation of Figure 3 describing the *PrvRed* procedure). First, consider the case when $K_{bct} \neq C_{trg}$. Then *Lrn* generates a certificate K as described in Subsection 7.2. Namely, it starts with K_{bct} gradually resolving out literals assigned at the conflict level by non-decision assignments. Since C_{trg} is not involved in derivation of K, the latter is a witness certificate.

Now, consider the case when $K_{bct} = C_{trg}$. If, no relevant assignment is derived from a witness certificate, *Lrn* generates the resulting certificate K as described above. Since C_{trg} is involved in derivation of K, the latter is a *participant* certificate that is added to the formula. If an assignment relevant to the conflict is derived from a witness certificate, *Lrn* acts differently. Namely, it derives a *witness* certificate K *and* a special clause $\hat{K}$ that is added to the formula. (For the sake of simplicity, we did not mention this fact in the pseudo-code of the *PrvRed* procedure.)

Figure 8 illustrates adding a special clause. Here $C_1 = \overline{x}_2 \vee x_3$ is the target clause. In the branch $y_1 = 0$, *PrvRed* proves C_1 redundant by deriving a witness certificate $K_1 = y_1 \vee \overline{x}_2$ (lines 9-13). Then *PrvRed* backtracks to level 0 and runs *BCP* to derive $y_1 = 1$ from K_1, $x_2 = 1$ from C_2 and $x_3 = 0$ from C_3. At this point, C_1 is falsified i.e. a conflict occurs. Assume we construct a certificate $K_2 = \overline{x}_2$ by resolving C_1 with C_2, C_3, and K_1 (i.e. with the clauses from which the relevant assignments were derived). Then we have a problem. On one hand, K_2 is a participant certificate that has to be added to F since the target clause C_1 was involved in building K_2. On the other hand, K_2

may not be implied by F since a witness certificate K_1 was involved in producing K_2. (A witness certificate K is, in general, only es-implied by F in $\exists X[K \wedge F]$.) This breaks the invariant maintained by *START* that only clauses implied by F are added to it.

The *Lrn* procedure addresses the problem above as follows. First, it generates a clause $\hat{K} = \overline{y}_1$ that is falsified in the current subspace and so "replaces" C_1 as the reason for the conflict. $\hat{K}$ is built without using witness certificates and so *can* be added to F. It is obtained by resolving C_1 with C_2 and C_3 and is added to F (lines 19-23). Then *Lrn* derives the certificate $K_2 = \overline{x}_2$ by resolving $\hat{K}$ and K_1. The clause K_2 certifies the redundancy of the target clause C_1 in the entire space. Note that K_2 was derived using $\hat{K}$ instead of the target clause C_1. So, it is a witness certificate that does not have to be added to the formula.

Here is how one handles the general case when $K_{bct} = C_{trg}$ and a witness certificate is involved in the conflict. First, one produces a special clause $\hat{K}$. It is obtained by resolving C_{trg} with clauses of F from which relevant assignments were derived. This process *stops* when the assignment derived from a *witness certificate* is reached. Then $\hat{K}$ is added to the formula F. (This can be done since witness certificates are not used in derivation of $\hat{K}$.) After that, *Lrn* generates a certificate K starting with $\hat{K}$ as a clause falsified in the current subspace. (That is, $\hat{K}$ replaces C_{trg} as the cause of the conflict.) Since C_{trg} is not involved in generation of K, the latter is a witness certificate.

E 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 $\exists X[F_1(X,Y) \wedge F_2(X,Y)]$. We will denote $F_1 \wedge F_2$ by F. In Subsection E.1, we show that *START* is sound. Subsection E.2 discusses the problem of generating duplicate clauses by *START* and describes a solution to this problem. In Subsection E.3, we show that the versions of *START* that do not produce duplicate clauses are complete.

E.1 *START* is sound

In its operation, *START* adds participant certificates and removes target clauses from F. Denote the initial formula F as F^{ini}. Let $\vec{y}$ be a full assignment to the variables of Y (i.e. unquantified ones). Below, we demonstrate that for every subspace $\vec{y}$, *START* preserves the equisatisfiability between the F^{ini} and the current formula F. That is, $\exists X[F^{ini}] \equiv \exists X[F]$. Then we use this fact to show that *START* produces a correct solution.

First, consider adding participant certificates by *START*. As we mention in Section 7, every clause added to F is *implied* by F. (If a clause K is es-implied by F in $\exists X[K \wedge F]$, it is used only as a *witness* certificate and is not added to F.) So, adding clauses cannot break equisatisfiability of F and F^{ini} in a subspace $\vec{y}$.

Now, we consider removing target clauses from F by *START*. A target clause C_{trg} is permanently removed from the formula only if a certificate K implying

C_{trg} in the entire space is derived. K is obtained by resolving clauses of the current formula F and witness certificates (if any). Derivation of K is correct due to correctness of Proposition 2 (describing generation of clauses that are es-implied rather than implied by the formula) and soundness of resolution. So, removing C_{trg} from F cannot break equisatisfiability of F and F^{ini} in some subspace $\vec{y}$.

The fact that $\exists X[F^{ini}] \equiv \exists X[F]$ entails that $START$ produces a correct solution. Indeed, $START$ terminates when the current formula F_1 does not contain a quantified clause. So, the *final* formula F can be represented as $F_1(Y) \wedge F_2(X, Y)$. Then $\exists X[F_1^{ini} \wedge F_2^{ini}] \equiv F_1 \wedge \exists X[F_2]$. $START$ does not add any clauses to F_2. Hence, the final and initial formulas F_2 are identical. So, $\exists X[F_1^{ini} \wedge F_2^{ini}] \equiv F_1^* \wedge \exists X[F_2^{ini}]$ where F_1^* is the final formula F_1.

E.2 Avoiding generation of duplicate clauses

The version of *PrvRed* described in Sections 6-8 may generate a duplicate of a quantified clause that is currently *proved redundant*. To avoid generating duplicates one can modify $START$ as follows. (We did not implement this modification due to its inefficiency. We present it just to show that the problem of duplicates can be fixed in principle.) We will refer to this modification as $START^*$.

Suppose *PrvRed* generated a quantified clause C proved redundant earlier. This can happen only when all variables of Y are assigned because they are assigned before those of X. Then $START^*$ discards the clause C, undoes the assignment to X, and eliminates all recursive calls of *PrvRed*. That is $START^*$ returns to the original call of *PrvRed* made in the main loop (Fig. 2, line 7). Let C_{trg} be the target clause of this call of *PrvRed* and $\vec{y}$ be the current (full) assignment to Y. At this point $START^*$ calls an internal SAT-solver to prove redundancy of C_{trg} in subspace $\vec{y}$. This goal is achieved by this SAT-solver via generating a witness or participant certificate implying C_{trg} in subspace $\vec{y}$ (see below). After that, *PrvRed* goes on as if it just finished line 10 of Figure 3.

Let $B(Y)$ denote the longest clause falsified by $\vec{y}$. Suppose the internal SAT-solver of $START^*$ proves $F_{\vec{y}}$ unsatisfiable. (Recall that F denotes $F_1 \wedge F_2$.) Then the clause B is a certificate of redundancy of C_{trg} in $F_{\vec{y}}$. If C_{trg} is involved in proving $F_{\vec{y}}$ unsatisfiable, B is a participant certificate. The *PrvRed* procedure adds B to F to make C_{trg} redundant in subspace $\vec{y}$. If $F_{\vec{y}}$ is proved unsatisfiable without using C_{trg}, then B is a witness certificate that is not added to F.

Suppose that $F_{\vec{y}}$ is satisfiable. Then the internal SAT-solver above derives an assignment $\vec{p}$ satisfying $F_{\vec{y}}$ where $\vec{y} \subseteq \vec{p}$. Note that $\vec{y}$ does not satisfy C_{trg} since, otherwise, *PrvRed* would have already proved redundancy of C_{trg} in subspace $\vec{y}$. Hence, $\vec{p}$ satisfies C_{trg} by an assignment to a variable $w \in X$. Then *PrvRed* derives a witness certificate K equal to $B \vee l(w)$ where $l(w)$ is the literal of w present in C_{trg}. It is not hard to show that K is indeed a certificate. First, it implies C_{trg} in subspace $\vec{y}$ certifying its redundancy there. Second, K is es-implied by $F \setminus \{C_{trg}\}$ in $\exists X[K \wedge (F \setminus \{C_{trg}\})]$.

E.3 *START* is complete

In this subsection, 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

- the number of times *START* calls the *PrvRed* procedure (to prove redundancy of the current target clause) is finite;
- the number of steps performed by one call of *PrvRed* is finite.

So, *START* always terminates. First, let us show that *PrvRed* is called a finite number of times. By our assumption, *START* does not generate quantified clauses seen before. So, the number of times *PrvRed* is called in the main loop of *START* (see Figure 2) 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 depth of recursion is finite here. Indeed, before a new recursive call is made, the unassigned variable $w \in X$ of C_{trg} is assigned and X is a finite set. Summarizing, the number of recursive calls made by *PrvRed* invoked in the main loop of *START* is finite.

Now we prove that the number of steps performed by a single call of *PrvRed* is finite. (Here we ignore the steps taken by *recursive* calls of *PrvRed*.) Namely, we show that *PrvRed* examines a finite search tree. The number of branching nodes of the search tree built by *PrvRed* is finite because $X \cup Y$ is a finite set. Let us show that *PrvRed* indeed builds a tree. That is *PrvRed* does not have "*holes*" and always reaches a *leaf* i.e. a node where a backtracking condition is met. Below, we list the four kinds of leafs reached by *PrvRed*. (The backtracking conditions are identified by the *BCP* procedure called by *PrvRed*.) Let $\vec{q}$ specify the current assignment to $Y \cup X$. A leaf of the first kind is reached when the target clause C_{trg} becomes unit in subspace $\vec{q}$. Then *BCP* calls the *Rcrs* procedure (line 11 of Fig. 4) and *PrvRed* backtracks. *PrvRed* reaches a leaf of the second kind when *BCP* finds a clause of F implying C_{trg} in subspace $\vec{q}$. A leaf of the third kind is reached when *BCP* identifies a clause falsified by $\vec{q}$ (i.e. a conflict occurs). *PrvRed* reaches a leaf of the fourth kind when the target clause C_{trg} is blocked in $\exists X[F]$ in subspace $\vec{q}$.

If F is unsatisfiable in subspace $\vec{q}$, *PrvRed* always reaches a leaf before all variables of $Y \cup X$ are assigned. (Assigning all variables without a conflict, i.e. without reaching a leaf of the third kind, would mean that F is satisfiable in subspace $\vec{q}$.) Let us show that if F is *satisfiable* in subspace $\vec{q}$, *PrvRed* also always reaches a leaf before every variable of $Y \cup X$ is assigned. (That is before a satisfying assignment is generated.) Let $\vec{p}$ be an assignment satisfying F where $\vec{q} \subseteq \vec{p}$. Consider the worst case scenario. That is all variables of $Y \cup X$ but some variable w are already assigned in $\vec{q}$ and no leaf condition is encountered yet. Assume that no literal of w is present in the target clause C_{trg}. Since $\vec{q}$ contains all assignments of $\vec{p}$ but that of w, C_{trg} is satisfied by $\vec{q}$. Recall that *PrvRed* does not make *decision* assignments satisfying C_{trg} (see Subsection 7.2). So, C_{trg} is satisfied by an assignment *derived* from a clause C. Then C implies

C_{trg} in subspace $\vec{q}$ and a leaf of the second kind must have been reached. So, we have a contradiction.

Now assume that C_{trg} has a literal $l(w)$ of w. Note that since *PrvRed* assigns variables of Y before those of X, then $w \in X$. Since C_{trg} is not implied by a clause of F in subspace $\vec{q}$, all the literals of C_{trg} but $l(w)$ are falsified by $\vec{q}$. Let us show that C_{trg} is blocked in $\exists X[F]$ at w in subspace $\vec{q}$. Assume the contrary i.e. there is a clause C resolvable with C_{trg} on w 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 $\vec{q}$. Then $\vec{p}$ cannot be a satisfying assignment because it falsifies either C_{trg} or C (depending on how the variable w is assigned). So, we have a contradiction.Thus, C_{trg} is blocked at w in subspace $\vec{q}$ and hence a leaf of the fourth kind is reached.

F Experiments With HWMCC-13 Benchmarks

In this appendix, we describe experiments with multi-property benchmarks of the HWMCC-13 set [38]. (We use this set because the multi-property track has been discontinued in HWMCC since 2013.) Each benchmark consists of a sequential circuit N and invariants that are supposed to hold for N. One can view the conjunction of those invariants as a **specification** Sp for N. In the experiments, we used $START$ to generate invariants of N not implied by Sp. Similarly to the experiment of Section 10, the formula $F_k = I(S_0) \wedge T(S_0,S_1) \wedge \cdots \wedge T(S_{k-1},S_k)$ was used to generate invariants. The number k of time frames was in the range of $2 \leq k \leq 10$. Specifically, we set k to the largest value in this range where $|F_k|$ did not exceed 500,000 clauses. We discarded the benchmarks with $|F_2| > 500,000$. So, in the experiments, we used 112 out of the 178 benchmarks of the set.

We describe three experiments. In every experiment, we generated local invariants H_k by taking out a clause of $\exists \mathbb{S}_{k-1}[F_k]$. The objective of the *first experiment* was to demonstrate that $START$ could compute H_k for realistic designs. We also showed in this experiment that PQE could be much easier than QE and that $START$ outperforms our previous PQE-solver called DS-PQE. The *second experiment* demonstrated that a clause Q of a local invariant H_k generated by $START$ was often a global invariant not implied by the specification Sp. (As we mentioned in Section 9, the necessary condition for an invariant Q to be unwanted is $Sp \not\Rightarrow Q$.) Note that the circuits of the HWMCC-13 set are "**anonymous**". So, we could not decide if Q was an unwanted invariant. Our goal was to show that $START$ was good enough to generate invariants not implied by Sp. (Then one could check those invariants for being unwanted as described in Section 11.) As in the experiment of Section 10, we took out only clauses containing a state variable of the k-th time frame. The choice of the next clause to take out was made according to the order in which clauses were listed in F_k. In the *third experiment*, we showed that $START$ generates invariants that are different from those produced by *IC3*.

F.1 Experiment 1

Table 2: *START* and *DS-PQE*. The time limit is 5 sec.

pqe solver	total probl.	solved	unsolved
start	5,418	**3,102**	**2,316**
ds-pqe	5,418	1,285	4,133

In this experiment, for each benchmark out of 112 mentioned above we generated PQE problems of taking a clause out of $\exists \mathbb{S}_{k-1}[F_k]$. Some of them were trivially solved by pre-processing. The latter eliminated the blocked clauses of F_k that could be easily identified and ran BCP launched due to the unit clauses specifying the initial state. We generated up to 50 *non-trivial* problems per benchmark ignoring those solved by pre-processing. (For some benchmarks the total number of non-trivial problems was under 50.)

We compared *START* with *DS-PQE* introduced in [1] that is based on the machinery of D-sequents. The relation of D-sequents and certificates is briefly discussed in Section 12. In contrast to *START*, *DS-PQE* proves redundancy of many targets *at once*, which can lead to generating very deep search trees. To make the experiment less time consuming, we limited the run time of *START* to 5 sec. per PQE problem. The results are shown in Table 2. The first column gives the name of a PQE solver. The second column shows the total number of PQE problems we generated for the 112 benchmarks. The last two columns give the number of problems solved and unsolved in the time limit. Table 2 shows that *START* solved 57% of the problems within 5 sec. For 92 benchmarks out of 112, at least one PQE problem generated off $\exists \mathbb{S}_{k-1}[F_k]$ was solved by *START* in the time limit. This is quite encouraging since many solved PQE problems had more than a hundred thousand variables and clauses. Table 2 also shows that *START* drastically outperforms *DS-PQE*.

To contrast PQE and QE, we used *CADET* [17,18] to perform QE on 112 formulas $\exists \mathbb{S}_{k-1}[F_k]$. That is, instead of taking a clause out of $\exists \mathbb{S}_{k-1}[F_k]$ by PQE, we applied *CADET* to perform full QE on this formula. (As mentioned in Subsection 9.2, performing QE on $\exists \mathbb{S}_{k-1}[F_k]$ produces the strongest local invariant specifying all states unreachable in k transitions.) Our choice of *CADET* was motivated by its high performance. *CADET* is a SAT-based tool that solves QE implicitly via building Skolem functions. In the context of QE, *CADET* often scales better than BDDs [21,39]. *CADET* solved only 32 out of 112 QE problems with the time limit of 600 sec. For many formulas $\exists \mathbb{S}_{k-1}[F_k]$ that *CADET* failed to solve in 600 sec., *START* solved all 50 PQE problems generated off $\exists \mathbb{S}_{k-1}[F_k]$ in 5 sec. So, PQE can be much easier than QE if only a small part of the formula gets unquantified.

F.2 Experiment 2

The second experiment was an extension of the first experiment. Namely, for each clause Q of a local invariant H_k generated by PQE we used *IC3* to verify if Q was a global invariant. If so, we checked if $Sp \not\Rightarrow Q$ held.

Table 3: A sample of HWMCC-13 benchmarks

name	lat-ches	invar. of *Sp*	time fra-mes	clau-ses taken out	local single-clause invariants				
					gen. inva-riants	global?			not impl. by *Sp*
						un-dec.	no	yes	
6s380	5,606	897	2	46	101	0	49	52	0
6s176	1,566	952	3	20	101	0	9	92	14
6s428	3,790	340	4	29	102	15	12	75	75
6s292	3,190	247	5	21	104	44	0	60	60
6s156	513	32	6	218	101	0	90	11	11
6s275	3,196	673	7	25	106	2	21	83	77
6s325	1,756	301	8	23	105	0	0	105	105
6s391	2,686	387	9	30	104	0	14	90	90
6s372	1,124	33	10	159	101	60	41	0	0

Similarly to the first experiment, to make the experiment less time consuming, we set the time limit of 5 sec. per PQE problem. Besides, we imposed the following constraints. (Even with those constraints, the run time of the experiment was about 4 days.) First, we stopped *START* even before the time limit if it generated more than 5 free clauses. Second, the time limit for *IC3* was set to 30 sec. Third, instead of constraining the number of PQE problems per benchmark, we limited the total number of free clauses generated for a benchmark. Namely, processing a benchmark terminated when this number exceeded 100.

A sample of 9 benchmarks out of the 112 we used in the experiment is shown in Table 3. Let us explain the structure of this table by the benchmark 6s380 (the first line of the table). The name of this benchmark is shown in the first column. The second column gives the number of latches (5,606). The number of invariants that should hold for 6s380 is provided in the third column (897). So, the specification *Sp* of 6s380 is the conjunction of those 897 invariants. The fourth column shows that the number k of time frames for 6s380 was set to 2 (since $|F_3| > 500,000$). The value 46 shown in the fifth column is the total number of clauses taken out of $\exists \mathbb{S}_{k-1}[F_k]$ i.e. the number of PQE problems. (We keep using the index k here assuming that $k\!=\!2$ for 6s380.)

Let C be a clause taken out of the scope of quantifiers by *START*. Every free clause Q generated by *START* was stored as a local single-clause invariant. The sixth column shows that taking clauses out of the scope of quantifiers was terminated when 101 local single-clause invariants were generated. (Because the total number of invariants exceeded 100.) Each of these 101 local invariants held in k-th time frame. The following three columns show how many of those 101 local invariants were true globally. *IC3* finished every problem out of 101 in the time limit. So, the number of undecided invariants was 0. The number of invariants *IC3* proved false or true globally was 49 and 52 respectively. The last column gives the number of global invariants *not* implied by *Sp*. For 6s380, this number is 0.

For 109 benchmarks out of the 112 we used in the experiments, *START* was able to generate local single-clause invariants that held in k-th time frame. For 100 benchmarks out of the 109 above, the invariants H_k generated by *START* contained global single-clause invariants. For 89 out of these 100 benchmarks, there were global invariants not implied by the specification *Sp*. Those invariants were meant to be checked if any of them was unwanted.

F.3 Experiment 3

When proving an invariant P, *IC3* conjoins it with clauses $Q_1,\ldots,Q_m$ to make $P \wedge Q_1 \wedge \cdots \wedge Q_m$ inductive. If *IC3* succeeds, every Q_i is an invariant. Moreover, Q_i may be an *unwanted* invariant. Arguably, the cause of efficiency of *IC3* is that P is often close to an inductive invariant. So, *IC3* needs to generate a relatively small number of clauses Q_i to make the constrained version of P inductive. However, as we mentioned in Section 12, this nice feature of *IC3* drastically limits the set of unwanted invariants it can produce. In this subsection, we substantiate this claim by an experiment. In this experiment, we picked the HWMCC-13 benchmarks for which one could prove *all* pre-defined invariants $P_1,\ldots,P_n$ within a time limit. Namely, for every benchmark we formed the specification $Sp = P_1 \wedge \cdots \wedge P_n$ and ran *IC3* to prove *Sp* true.

We selected the benchmarks that *IC3* solved in less than 1000 sec. (In addition to dropping the benchmarks not solved in 1000 sec., we discarded those where *Sp* failed because some invariants P_i were false). Let $\boldsymbol{Sp^*}$ denote the inductive version of *Sp* produced by *IC3* when proving *Sp* true. That is, Sp^* is *Sp* conjoined with the invariant clauses generated by *IC3*. For each of the selected benchmarks we generated invariants by *START* exactly as in Experiment 2. That is, we stopped generation of local single clause invariants when their number exceeded 100. Then we ran *IC3* to identify local invariants that were global as well. After that we checked which of the global invariants generated by *START* were not implied by *Sp* and Sp^*.

Table 4: Invariants of *START* and *IC3*

name	latches	invariants in *Sp*	glob sngl cls invars		
			glob. invariants	not impl. by *Sp*	not impl. by Sp^*
6s135	2,307	340	68	68	61
6s325	1,756	301	101	101	96
ex1	130	33	29	21	19
ex2	212	32	93	61	42
6s106	135	17	100	86	83
6s256	3,141	5	0	0	0
ex3	61	3	2	2	2
ex4	63	3	3	3	3
6s209	5,759	2	73	72	66
6s113	994	1	18	17	17
6s143	260	1	103	83	77
6s170	3,141	1	1	1	1
6s252	170	1	94	71	65
Total				**586**	**532**

The results of the experiment are shown in Table 4. The first three columns of this table are the same as in Table 3. They give the name of a benchmark, the number of latches and the number of invariants $P_1,\ldots,P_n$ to prove. (The actual names of examples *ex1*,..,*ex4* in the HWMCC-13 set are *pdtvsarmultip*, *bobtuintmulti*, *nusmvdme1d3multi*, *nusmvdme2d3multi*.) The next two columns of Table 4 are the same as the last two columns of Table 3. They show the number of local invariant clauses that turn out to be global invariants and the number of global invariants that were not implied by *Sp*. The last column gives the number of global invariants that were not implied by Sp^*. The last row of the table shows that in 532 cases out of 586 the invariants not implied by *Sp* were not implied by Sp^* either. So, in 90% of cases *START* generated invariant clauses *different* from those of *IC3*.