Partial Quantifier Elimination And Property Generation

Eugene Goldberg

eu.goldberg@gmail.com

Abstract. We study partial quantifier elimination (PQE) for propositional CNF formulas with existential quantifiers. 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 clauses. The appeal of PQE is that many verification problems (e.g. equivalence checking and model checking) can be solved in terms of PQE and the latter can be dramatically simpler than full quantifier elimination. We show that PQE can be used for property generation that can be viewed as a generalization of testing. The objective here is to produce an *unwanted* property of a design implementation thus exposing a bug. We introduce two PQE solvers called *EG-PQE* and *EG-PQE*$^+$. *EG-PQE* is a very simple SAT-based algorithm. *EG-PQE*$^+$ is more sophisticated and robust than *EG-PQE*. We use these PQE solvers to find an unwanted property (namely, an unwanted invariant) of a buggy FIFO buffer. We also apply them to invariant generation for sequential circuits from a HWMCC benchmark set. Finally, we use these solvers to generate properties of a combinational circuit that mimic symbolic simulation.

1 Introduction

In this paper, we consider the following problem. Let $F(X,Y)$ be a propositional formula in conjunctive normal form (CNF)[1] where X, Y are sets of variables. Let G be a subset of clauses of F. Given a formula $\exists X[F]$, find a quantifier-free formula $H(Y)$ such that $\exists X[F] \equiv H \wedge \exists X[F \setminus G]$. In contrast to *full* quantifier elimination (QE), only the clauses of G are taken out of the scope of quantifiers here. So, this problem is called *partial* QE (PQE) [1]. (In this paper, we consider PQE only for formulas with *existential* quantifiers.) We will refer to H 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 $G = F$ and the entire formula is unquantified. Interpolation [2,3] is a special case of PQE as well [4] (see also Appendix A).

The appeal of PQE is threefold. First, it can be much more efficient than QE if G is a *small* subset of F. Second, many verification problems like SAT,

[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 $\{C_1, \ldots, C_k\}$.

equivalence checking, model checking can be solved in terms of PQE [1,5,6,7]. So, PQE can be used to design new efficient methods for solving known problems. Third, one can apply PQE to solving *new* problems like property generation considered in this paper. In practice, to perform PQE, it suffices to have an algorithm that takes a single clause out of the scope of quantifiers. Namely, given a formula $\exists X[F(X,Y)]$ and a clause $C \in F$, this algorithm finds a formula $H(Y)$ such that $\exists X[F] \equiv H \wedge \exists X[F \setminus \{C\}]$. To take out k clauses, one needs to apply this algorithm k times. Since $H \wedge \exists X[F] \equiv H \wedge \exists X[F \setminus \{C\}]$, solving the PQE above reduces to finding $H(Y)$ that makes C *redundant* in $H \wedge \exists X[F]$. So, the PQE algorithms we present here are based on *redundancy based reasoning*.

We describe two PQE algorithms called *EG-PQE* and *EG-PQE*$^+$ where "*EG*" stands for "Enumerate and Generalize". *EG-PQE* is a very simple SAT-based algorithm that can sometimes solve very large problems. *EG-PQE*$^+$ is a modification of *EG-PQE* that makes the algorithm more powerful and robust[2]. To show the practicality of PQE we apply it to property generation. Our motivation here is as follows. In practice, the set of properties a design must meet is incomplete. (That is, this design can be buggy even if all properties of this set hold.) This problem is usually addressed by massive testing. The input/output behavior of the design under a single test can be cast as a (simple) design property. So, a test exposing a bug can be viewed as identifying an *unwanted* design property. In terms of property generation, the flaw of testing is that it considers only very simple properties.

In this paper, we show that one can use PQE to generate more complex design properties. The goal of property generation is to produce an unwanted property thus exposing a bug that may have been overlooked or simply *cannot* be detected by testing. The benefits of property generation by PQE are as follows. First, by using PQE one can make property generation efficient. Second, PQE facilitates generation of properties covering different parts of the design, which increases the probability of discovering a bug. Third, every property generated by PQE specifies a large set of high-quality tests. We show how PQE can be used to generate properties for a combinational circuit. We also continue the work on *invariant generation* for a sequential circuit we started in [8]. One can use invariant generation to identify a bug that makes a state of a sequential circuit *unreachable* (whereas in a correct design this state should be reachable). Such bugs can be easily missed.

The main body of this paper is structured as follows. (A full version with all appendices will be published as a technical report.) In Section 2, we give basic definitions. Section 3 presents property generation for a combinational circuit. In Section 4, we describe invariant generation for a sequential circuit. Sections 5 and 6 present *EG-PQE* and *EG-PQE*$^+$ respectively. Section 7 makes some remarks about experiments and relates our previous PQE-solver called *START* [8] to *EG-PQE*$^+$. In Section 8, invariant generation is used to find a

[2] Earlier we introduced a PQE-solver called *START* [8]. In this paper, we reproduce some experiments of [8] conducted with *START* using *EG-PQE* and *EG-PQE*$^+$. The relation between *START*, *EG-PQE* and *EG-PQE*$^+$ is explained in Section 7.

hard bug in a FIFO buffer. Experiments with invariant generation for HWMCC benchmarks are described in Section 9. Section 10 presents an experiment with property generation for combinational circuits. In Sections 11 and 12, we give some background and make conclusions.

2 Basic Definitions

In this section, when we say "formula" without mentioning quantifiers, we mean "a quantifier-free formula".

Definition 1. *We assume that formulas have only Boolean variables. A* ***literal*** *of a variable v is either v or its negation. A* ***clause*** *is a disjunction of literals. A formula F is in conjunctive normal form (**CNF**) if $F = C_1 \wedge \cdots \wedge C_k$ where $C_1, \ldots, C_k$ are clauses. We will also view F as a* ***set of clauses*** *$\{C_1, \ldots, C_k\}$. We assume that* ***every formula is in CNF*** *unless otherwise stated.*

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

Definition 3. *Let V be a set of variables. An* ***assignment*** $\vec{q}$ *to V is a mapping $V' \rightarrow \{0,1\}$ where $V' \subseteq V$. We will denote the set of variables assigned in $\vec{q}$ as* $\boldsymbol{Vars(\vec{q})}$*. 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 4. *A literal and a clause are said to be* ***satisfied*** *(respectively* ***falsified****) by an assignment $\vec{q}$ if they evaluate to 1 (respectively 0) under $\vec{q}$.*

Definition 5. *Let C be a clause. 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 6. *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 7. *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 8. *Let $F(X,Y)$ be a formula and $G \subseteq F$ and $G \neq \emptyset$. The clauses of G are said to be* $\boldsymbol{redundant\ in\ \exists X[F]}$ *if $\exists X[F] \equiv \exists X[F \setminus G]$. Note that if $F \setminus G$ implies G, the clauses of G are redundant in $\exists X[F]$, however, the converse is not true.*

Definition 9. *Given a formula $\exists X[F(X,Y))]$ and G where $G \subseteq F$, the* ***Partial Quantifier Elimination*** *(**PQE**) problem is to find $H(Y)$ such that $\exists \boldsymbol{X}[\boldsymbol{F}] \equiv \boldsymbol{H} \wedge \exists \boldsymbol{X}[\boldsymbol{F} \setminus \boldsymbol{G}]$. (So, PQE takes G out of the scope of quantifiers.) The formula H is called a* ***solution*** *to PQE. The case of PQE where $G = F$ is called* ***Quantifier Elimination*** *(**QE**).*

Remark 1. Let D be a clause of a solution H to the PQE problem of Definition 9. If $F \setminus G$ implies D, then $H \setminus \{D\}$ is a solution to this PQE problem too.

Example 1. Consider the formula $F = C_1 \wedge C_2 \wedge C_3 \wedge C_4$ where $C_1 = \overline{x}_3 \vee x_4$, $C_2 = y_1 \vee x_3$, $C_3 = y_1 \vee \overline{x}_4$, $C_4 = y_2 \vee x_4$. Let Y denote $\{y_1, y_2\}$ and X denote $\{x_3, x_4\}$. Consider the PQE problem of taking C_1 out of $\exists X[F]$ i.e. finding $H(Y)$ such that $\exists X[F] \equiv H \wedge \exists X[F \setminus \{C_1\}]$. As we show later, $\exists X[F] \equiv y_1 \wedge \exists X[F \setminus \{C_1\}]$. That is, $H = y_1$ is a solution to the PQE problem above.

Proposition 1. *Let H be a solution to the PQE problem of Definition 9. That is $\exists X[F] \equiv H \wedge \exists X[F \setminus G]$. Then $F \Rightarrow H$ (i.e. F implies H).*

The proofs of propositions are given in Appendix B.

Definition 10. *Let clauses C',C'' have opposite literals of exactly one variable $w \in Vars(C') \cap Vars(C'')$. 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.*

Definition 11. *Let C be a clause of a formula G and $w \in Vars(C)$. The clause C is said to be* ***blocked*** *[9] in G with respect to the variable w if no clause of G is resolvable with C on w.*

Proposition 2. *Let a clause C be blocked in a formula $F(X,Y)$ with respect to a variable $x \in X$. Then C is redundant in $\exists X[F]$ i.e. $\exists X[F] \equiv \exists X[F \setminus \{C\}]$.*

3 Property Generation By PQE

Many known problems can be formulated in terms of PQE thus facilitating the design of new efficient algorithms. In Appendix C, we recall some results on solving SAT, equivalence checking and model checking by PQE presented in [1,5,6,7]. In this section, we describe application of PQE to *property generation* for a combinational circuit. The objective of property generation is to expose a bug via producing an *unwanted* property.

Let $M(X,V,W)$ be a combinational circuit where X, V, W specify the sets of the internal, input, and output variables of M respectively. Let $F(X,V,W)$ denote a formula specifying M. As usual, this formula is obtained by Tseitin's transformations [10]. Namely, F equals $F_{G_1} \wedge \cdots \wedge F_{G_k}$ where $G_1, \ldots, G_k$ are the gates of M and F_{G_i} specifies the functionality of gate G_i.

Example 2. Let G be a 2-input AND gate defined as $x_3 = x_1 \wedge x_2$ where x_3 denotes the output value and x_1, x_2 denote the input values of G. Then G is specified by the formula $F_G = (\overline{x}_1 \vee \overline{x}_2 \vee x_3) \wedge (x_1 \vee \overline{x}_3) \wedge (x_2 \vee \overline{x}_3)$. Every clause of F_G is falsified by an inconsistent assignment (where the output value of G is not implied by its input values). For instance, $x_1 \vee \overline{x}_3$ is falsified by the inconsistent assignment $x_1 = 0, x_3 = 1$. So, every assignment *satisfying* F_G corresponds to a *consistent* assignment to G and vice versa. Similarly, every assignment satisfying the formula F above is a consistent assignment to the gates of M and vice versa.

3.1 Property generation as generalization of testing

Let $w_i \in W$ be an output variable of M and $\vec{v}$ be a test i.e. a full assignment to the input variables V of M. Let $B^{\vec{v}}$ denote the longest clause falsified by $\vec{v}$ i.e. $Vars(B^{\vec{v}}) = V$. Let $l(w_i)$ be the literal satisfied by the value of w_i produced by M under input $\vec{v}$. Then the clause $B^{\vec{v}} \vee l(w_i)$ is satisfied by every assignment satisfying F i.e. $B^{\vec{v}} \vee l(w_i)$ is a property of M. We will refer to it as a **single-test property** (since it describes the behavior of M for a single test). If the input $\vec{v}$ is supposed to produce the opposite value of w_i (i.e. the one *falsifying* $l(w_i)$), then $\vec{v}$ exposes a bug in M. In this case, the single-test property above is an **unwanted** property of M exposing the same bug as the test $\vec{v}$.

A single-test property can be viewed as a weakest property of M as opposed to the strongest property specified by $\exists X[F]$. The latter is the truth table of M that can be explicitly computed by performing QE on $\exists X[F]$. One can use PQE to generate properties of M that, in terms of strength, range from the weakest ones to the strongest property inclusively. (By combining clause splitting with PQE one can generate single-test properties, see the next subsection.) Consider the PQE problem of taking a clause C out of $\exists X[F]$. Let $H(V, W)$ be a solution to this problem i.e. $\exists X[F] \equiv H \wedge \exists X[F \setminus \{C\}]$. Since H is implied by F, it can be viewed as a **property** of M. If H is an **unwanted** property, M has a bug. (Here we consider the case where a property of M is obtained by taking a clause out of formula $\exists X[F]$ where only the *internal* variables of M are quantified. Later we consider cases where some external variables of M are quantified too.)

The benefit of property generation by PQE is fourfold. First, by property generation one can identify bugs that are hard or simply impossible to find by testing. Second, using PQE makes property generation efficient. Third, by taking out different clauses one can generate properties covering different parts of design. This increases the probability of discovering a bug. Fourth, every property generated by PQE specifies a large set of high-quality tests.

We will assume that the property H generated by PQE has no redundant clauses (see Remark 1). That is if $D \in H$, then $F \setminus \{C\} \not\Rightarrow D$. Then one can view H as a property that holds due to the presence of the clause C in F.

3.2 Computing properties efficiently

If a property H is obtained by taking only one clause out of $\exists X[F]$, its computation is much easier than performing QE on $\exists X[F]$. If computing H still

remains too time-consuming, one can use the two methods below that achieve better performance at the expense of generating weaker properties. The first method applies when a PQE solver forms a solution *incrementally*, clause by clause (like the algorithms described in Sections 5 and 6). Then one can simply stop computing H as soon as the number of clauses in H exceeds a threshold. Since an "incomplete" formula H is implied by F, it specifies a property of M.

The second method employs *clause splitting*. Here we consider clause splitting on input variables $v_1, \ldots, v_p$ i.e. those of V (but one can split a clause on any subset of variables from $Vars(F)$). Let F' denote the formula F where a clause C is replaced with $p+1$ clauses: $C_1 = C \vee \overline{l(v_1)}, \ldots, C_p = C \vee \overline{l(v_p)}$, $C_{p+1} = C \vee l(v_0) \vee \cdots \vee l(v_p)$, where $l(v_i)$ is a literal of v_i. The idea is to obtain a property H by taking the clause C_{p+1} out of $\exists X[F']$ rather than C out of $\exists X[F]$. The former PQE problem is simpler than the latter since it produces a weaker property H. One can show that if $\{v_1, \ldots, v_p\} = V$, then a) the complexity of PQE reduces to **linear**; b) taking out C_{p+1} actually produces a **single-test property**. The latter specifies the input/output behavior of M for the test $\vec{v}$ falsifying the literals $l(v_1), \ldots, l(v_p)$. (See Appendix D for more details.)

3.3 Using design coverage for generation of unwanted properties

Arguably, testing is so effective in practice because one verifies a *particular design*. Namely, one probes different parts of this design using some coverage metric rather than samples the truth table (which would mean verifying *every possible design*). The same idea works for property generation by PQE for the following two reasons. First, by taking out a clause, PQE generates a property inherent to the *specific* circuit M. (If one replaces M with an equivalent but structurally different circuit, PQE will generate different properties.) Second, by taking out different clauses of F one generates properties corresponding to different parts of M thus "covering" the design. This increases the chance to take out a clause corresponding to the buggy part of M and generate an unwanted property.

3.4 High-quality tests specified by a property generated by PQE

In this subsection, we show that a property H generated by PQE, in general, specifies a large set of high-quality tests. Let $H(V, W)$ be obtained by taking C out of $\exists X[F(X, V, W)]$. Let $Q(V, W)$ be a clause of H. As mentioned above, we assume that $F \setminus \{C\} \not\Rightarrow Q$. Then there is an assignment $(\vec{x}, \vec{v}, \vec{w})$ satisfying formula $(F \setminus \{C\}) \wedge \overline{Q}$ where $\vec{x}, \vec{v}, \vec{w}$ are assignments to X, V, W respectively. (Note that by definition, $(\vec{v}, \vec{w})$ falsifies Q.) Let $(\vec{x}^*, \vec{v}, \vec{w}^*)$ be the execution trace of M under the input $\vec{v}$. So, $(\vec{x}^*, \vec{v}, \vec{w}^*)$ satisfies F. Note that the output assignments $\vec{w}$ and $\vec{w}^*$ must be different because $(\vec{v}, \vec{w}^*)$ has to satisfy Q. (Otherwise, $(\vec{x}^*, \vec{v}, \vec{w}^*)$ satisfies $F \wedge \overline{Q}$ and so $F \not\Rightarrow Q$ and hence $F \not\Rightarrow H$.) So, one can view $\vec{v}$ as a test "detecting" disappearance of the clause C from F. Note that different assignments satisfying $(F \setminus \{C\}) \wedge \overline{Q}$ correspond to different tests $\vec{v}$. So, the clause Q of H, in general, specifies a very large number of tests. One can show that these tests are similar to those detecting stuck-at faults and so have very high quality (see Appendix E for more details).

3.5 Identifying unwanted properties

In some cases, e.g. those mentioned in Sections 8 and 10, it is easy to decide if a property H obtained by PQE is unwanted. Otherwise, this can be done by analyzing tests detecting the disappearance of a clause from F (see Subsection 3.4). A more detailed discussion of this topic is beyond the scope of this paper.

4 Invariant Generation By PQE

In [8], we showed that PQE can be used for generating *invariants* of a sequential circuit. In this paper, we continue this topic in a more general context of property generation. We use generation of invariants (over that of weaker properties just claiming that a state cannot be reached in k transitions or less) because identification of an *unwanted* invariant is, arguably, easier. This simplifies bug detection by property generation.

4.1 Bugs making states unreachable

Let N be a sequential circuit and S denote the state variables of N. Let $I(S)$ specify the initial state $\vec{s}_{ini}$ (i.e. $I(\vec{s}_{ini})=1$). Let $T(S',V,S'')$ denote the transition relation of N where S', S'' are the present and next state variables and V specifies the (combinational) input variables. We will say that a state $\vec{s}$ of N is reachable if there is an execution trace leading to $\vec{s}$. That is, there is a sequence of states $\vec{s_0},\ldots,\vec{s_k}$ where $\vec{s_0} = \vec{s}_{ini}$, $\vec{s_k} = \vec{s}$ and there exist $\vec{v}_i$ $i=0,\ldots,k-1$ for which $T(\vec{s}_i,\vec{v}_i,\vec{s}_{i+1}) = 1$. Let N have to satisfy a set of **invariants** $P_0(S),\ldots,P_m(S)$. That is P_i holds iff $P_i(\vec{s}) = 1$ for every reachable state $\vec{s}$ of N. We will denote the **aggregate invariant** $P_0 \wedge \cdots \wedge P_m$ as $\boldsymbol{P_{agg}}$. We will call $\vec{s}$ a **bad state** of N if $P_{agg}(\vec{s}) = 0$. If P_{agg} holds, no bad state is reachable. We will call $\vec{s}$ a **good state** of N if $P_{agg}(\vec{s}) = 1$.

Typically, the set of invariants $P_0,\ldots,P_m$ is incomplete in the sense that it does not specify all states that must be *unreachable*. So, a good state can well be unreachable. We will call a good state **operative** (or **op-state** for short) if it is supposed to be used by N and so should be *reachable*. We introduce the term *an operative state* just to factor out "useless" good states. We will say that N has an **op-state reachability bug** if an op-state is unreachable in N. In Section 8, we consider such a bug in a FIFO buffer. The fact that P_{agg} holds says *nothing* about reachability of op-states. Consider, for instance, a trivial circuit N_{triv} that simply stays in the initial state $\vec{s}_{ini}$ and $P_{agg}(\vec{s}_{ini}) = 1$. Then P_{agg} holds for N_{triv} but the latter has op-state reachability bugs (assuming that the correct circuit must reach states other than $\vec{s}_{ini}$).

Let $R_{\vec{s}}$ be the predicate satisfied only by a state $\vec{s}$. In terms of CTL, identifying an op-state reachability bug means finding $\vec{s}$ for which the property $EF.R_{\vec{s}}$ must hold but it does not. The reason for assuming $\vec{s}$ to be *unknown* is that the set of op-states is typically too large to *explicitly* specify every property $ET.R_{\vec{s}}$ to hold. This makes finding op-state reachability bugs very hard.

The problem is exacerbated by the fact that reachability of different states is established by *different traces*. So, in general, one cannot efficiently prove many properties $EF.R_{\vec{s}}$ (for different states) *at once*.

4.2 Proving op-state unreachability by invariant generation

In practice, there are two methods to check reachability of operative states for large circuits. The first method is testing. Of course, testing cannot prove a state unreachable, however, the examination of execution traces may point to a potential problem. (For instance, after examining execution traces of the circuit N_{triv} above one realizes that many operative states look unreachable.) The other method is to check **unwanted invariants** i.e. those that are supposed to fail. If an unwanted invariant holds for a circuit, then this circuit has an op-state reachability bug. For instance, one may check if a state variable $s_i \in S$ of a circuit never changes its initial value. To break this unwanted invariant, one needs to find an operative state where the initial value of s_i is flipped. (For the circuit N_{triv} above this unwanted invariant holds for every state variable.) The potential unwanted invariants are formed manually i.e. simply *guessed*.

The two methods above can easily overlook an op-state reachability bug. Testing cannot prove that an op-state is unreachable. To correctly guess an unwanted invariant that holds, one essentially has to know the underlying bug. Below, we describe a method for invariant generation by PQE that is based on property generation for combinational circuits. The appeal of this method is twofold. First, PQE generates invariants "inherent" to the implementation at hand, which drastically reduces the set of invariants to explore. Second, PQE is able to generate invariants related to different parts of the circuit (including the buggy one). This increases the probability of generating an unwanted invariant. We substantiate this claim in Section 8.

Let formula $\boldsymbol{F_k}$ specify the combinational circuit obtained by unfolding a sequential circuit N for k time frames and adding the initial state constraint $I(S_0)$. That is $F_k = I(S_0) \wedge T(S_0, V_0, S_1) \wedge \cdots \wedge T(S_{k-1}, V_{k-1}, S_k)$ where S_j, V_j denote the state and input variables of j-th time frame respectively, $0 \leq j \leq k$. Let $H(S_k)$ be a solution to the PQE problem of taking a clause C out of $\exists X_k[F_k]$ where $X_k = S_0 \cup V_0 \cup \cdots \cup S_{k-1} \cup V_{k-1}$. That is $\exists X_k[F_k] \equiv H \wedge \exists X_k[F_k \setminus \{C\}]$. Note that in contrast to Section 3, here some external variables of the combinational circuit (namely, the input variables $V_0, \ldots, V_{k-1}$) are quantified too. So, H depends only on state variables of the last time frame. H can be viewed as a **local invariant** asserting that no state falsifying H can be reached in k transitions.

One can use H to find global invariants (holding for *every* time frame) as follows. Even if H is only a local invariant, a clause Q of H can be a *global* invariant. The experiments of Section 9 show that, in general, this is true for many clauses of H. (To find out if Q is a global invariant, one can simply run a model checker to see if the property Q holds.) Note that by taking out different clauses of F_k one can produce global single-clause invariants Q relating to dif-

ferent parts of N. From now on, when we say "an invariant" without a qualifier we mean a **global invariant** that holds in every time frame.

5 Introducing *EG-PQE*

In this section, we describe a simple SAT-based algorithm for performing PQE called *EG-PQE*. Here *'EG'* stands for 'Enumerate and Generalize'. *EG-PQE* accepts a formula $\exists X[F(X,Y)]$ and a clause $C \in F$. It outputs a formula $H(Y)$ such that $\exists X[F_{ini}] \equiv H \wedge \exists X[F_{ini} \setminus \{C\}]$ where F_{ini} is the initial formula F. (This point needs clarification because *EG-PQE* changes F by adding clauses.)

5.1 An example

Before describing the pseudocode of *EG-PQE*, we explain how it solves the PQE problem of Example 1. That is we consider taking clause C_1 out of $\exists X[F(X,Y)]$ where $F = C_1 \wedge \cdots \wedge C_4$, $C_1 = \overline{x}_3 \vee x_4$, $C_2 = y_1 \vee x_3$, $C_3 = y_1 \vee \overline{x}_4$, $C_4 = y_2 \vee x_4$ and $Y = \{y_1, y_2\}$ and $X = \{x_3, x_4\}$.

EG-PQE iteratively generates a full assignment $\vec{y}$ to Y and checks if $(C_1)_{\vec{y}}$ is redundant in $\exists X[F_{\vec{y}}]$ (i.e. if C_1 is redundant in $\exists X[F]$ in subspace $\vec{y}$). Note that if $(F \setminus \{C_1\})_{\vec{y}}$ *implies* $(C_1)_{\vec{y}}$, then $(C_1)_{\vec{y}}$ is trivially redundant in $\exists X[F_{\vec{y}}]$. To avoid such subspaces, *EG-PQE* generates $\vec{y}$ by searching for an assignment $(\vec{y}, \vec{x})$ satisfying the formula $(F \setminus \{C_1\}) \wedge \overline{C}_1$. (Here $\vec{y}$ and $\vec{x}$ are full assignments to Y and X respectively.) If such $(\vec{y}, \vec{x})$ exists, it satisfies $F \setminus \{C_1\}$ and falsifies C_1 thus proving that $(F \setminus \{C_1\})_{\vec{y}}$ *does not* imply $(C_1)_{\vec{y}}$.

Assume that *EG-PQE* found an assignment $(y_1 = 0, y_2 = 1, x_3 = 1, x_4 = 0)$ satisfying $(F \setminus \{C_1\}) \wedge \overline{C}_1$. So $\vec{y} = (y_1 = 0, y_2 = 1)$. Then *EG-PQE* checks if $F_{\vec{y}}$ is satisfiable. $F_{\vec{y}} = (\overline{x}_3 \vee x_4) \wedge x_3 \wedge \overline{x}_4$ and so it is *unsatisfiable*. This means that $(C_1)_{\vec{y}}$ *is not* redundant in $\exists X[F_{\vec{y}}]$. (Indeed, $(F \setminus \{C_1\})_{\vec{y}}$ is satisfiable. So, removing C_1 makes F satisfiable in subspace $\vec{y}$.) *EG-PQE makes* $(C_1)_{\vec{y}}$ redundant in $\exists X[F_{\vec{y}}]$ by **adding** to F a clause B falsified by $\vec{y}$. The clause B equals y_1 and is obtained by identifying the assignments to individual variables of Y that made $F_{\vec{y}}$ unsatisfiable. (In our case, this is the assignment $y_1 = 0$.) Note that derivation of clause y_1 *generalizes* the proof of unsatisfiability of F in subspace $(y_1 = 0, y_2 = 1)$ so that this proof holds for subspace $(y_1 = 0, y_2 = 0)$ too.

Now *EG-PQE* looks for a new assignment satisfying $(F \setminus \{C_1\}) \wedge \overline{C}_1$. Let the assignment $(y_1 = 1, y_2 = 1, x_3 = 1, x_4 = 0)$ be found. So, $\vec{y} = (y_1 = 1, y_2 = 1)$. Since $(y_1 = 1, y_2 = 1, x_3 = 0)$ satisfies F, the formula $F_{\vec{y}}$ is satisfiable. So, $(C_1)_{\vec{y}}$ is *already redundant* in $\exists X[F_{\vec{y}}]$. To avoid re-visiting the subspace $\vec{y}$, *EG-PQE* generates the **plugging** clause $D = \overline{y}_1 \vee \overline{y}_2$ falsified by $\vec{y}$.

EG-PQE fails to generate a new assignment $\vec{y}$ because the formula $D \wedge (F \setminus \{C_1\}) \wedge \overline{C}_1$ is unsatisfiable. Indeed, every full assignment $\vec{y}$ we have examined so far falsifies either the clause y_1 added to F or the plugging clause D. The only assignment *EG-PQE* has not explored yet is $\vec{y} = (y_1 = 1, y_2 = 0)$. Since $(F \setminus \{C_1\})_{\vec{y}} = x_4$ and $(C_1)_{\vec{y}} = \overline{x}_3 \vee x_4$, the formula $(F \setminus \{C_1\}) \wedge \overline{C}_1$ is

unsatisfiable in subspace $\vec{y}$. In other words,$(C_1)_{\vec{y}}$ is implied by $(F \setminus \{C_1\})_{\vec{y}}$ and hence is redundant. Thus, C_1 is redundant in $\exists X[F_{ini} \wedge y_1]$ for every assignment to Y where F_{ini} is the initial formula F. That is, $\exists X[F_{ini}] \equiv y_1 \wedge \exists X[F_{ini} \setminus \{C_1\}]$ and so the clause y_1 is a solution H to our PQE problem.

5.2 Description of *EG-PQE*

The pseudo-code of *EG-PQE* is shown in Fig. 1. *EG-PQE* starts with storing the initial formula F and initializing formula Plg that accumulates the plugging clauses generated by *EG-PQE* (line 1). As we mentioned in the previous subsection, plugging clauses are used to avoid re-visiting the subspaces where the formula F is proved satisfiable.

All the work is carried out in a while loop. First, *EG-PQE* checks if there is a new subspace $\vec{y}$ where $\exists X[(F \setminus \{C\})_{\vec{y}}]$ does not imply $F_{\vec{y}}$. This is done by searching for an assignment $(\vec{y}, \vec{x})$ satisfying $Plg \wedge (F \setminus \{C\}) \wedge \overline{C}$ (lines 3-4). If such an assignment does not exist, the clause C is redundant in $\exists X[F]$. (Indeed, let $\vec{y}$ be a full assignment to Y. The formula $Plg \wedge (F \setminus \{C\}) \wedge \overline{C}$ is unsatisfiable in subspace $\vec{y}$ for one of the two reasons. First, $\vec{y}$ falsifies Plg. Then $C_{\vec{y}}$ is redundant because $F_{\vec{y}}$ is satisfiable. Second, $(F \setminus \{C\})_{\vec{y}} \wedge \overline{C_{\vec{y}}}$ is unsatisfiable. In this case, $(F \setminus \{C\})_{\vec{y}}$ implies $C_{\vec{y}}$.) Then *EG-PQE* returns the set of clauses added to the initial formula F as a solution H to the PQE problem (lines 5-6).

```
EG-PQE(F, X, Y, C) {
1   Plg := ∅; F_ini := F
2   while (true) {
3     G := F \ {C}
4     y⃗ := Sat_1(Plg ∧ G ∧ ¬C)
5     if (y⃗ = nil)
6       return(F \ F_ini)
7     (x⃗*, B) := Sat_2(F, y⃗)
8     if (B ≠ nil) {
9       F := F ∪ {B}
10      continue }
11    D := PlugCls(y⃗, x⃗*, F)
12    Plg := Plg ∪ {D}}}
```

Fig. 1: Pseudocode of *EG-PQE*

If the satisfying assignment $(\vec{y}, \vec{x})$ above exists, *EG-PQE* checks if the formula $F_{\vec{y}}$ is satisfiable (line 7). If not, then the clause $C_{\vec{y}}$ *is not* redundant in $\exists X[F_{\vec{y}}]$ (because $(F \setminus \{C\})_{\vec{y}}$ is satisfiable). So, *EG-PQE makes* $C_{\vec{y}}$ redundant by generating a clause $B(Y)$ falsified by $\vec{y}$ and adding it to F (line 9). Note that adding B also prevents *EG-PQE* from re-visiting the subspace $\vec{y}$ again. The clause B is built by finding an *unsatisfiable* subset of $F_{\vec{y}}$ and collecting the literals of Y removed from clauses of this subset when obtaining $F_{\vec{y}}$ from F. The unsatisfiable subset above can be easily extracted from a resolution proof that $F_{\vec{y}}$ is unsatisfiable.

If $F_{\vec{y}}$ is satisfiable, *EG-PQE* generates an assignment $\vec{x}^*$ to X such that $(\vec{y}, \vec{x}^*)$ satisfies F (line 7). The satisfiability of $F_{\vec{y}}$ means that every clause of $F_{\vec{y}}$ including $C_{\vec{y}}$ is redundant in $\exists X[F_{\vec{y}}]$. At this point, *EG-PQE* uses the longest clause $D(Y)$ falsified by $\vec{y}$ as a plugging clause (line 11). The clause D is added to Plg to avoid re-visiting subspace $\vec{y}$. Sometimes it is possible to remove variables from $\vec{y}$ to produce a shorter assignment $\vec{y}^*$ such that $(\vec{y}^*, \vec{x}^*)$ still satisfies F. Then one can use a shorter plugging clause D involving only the variables assigned in $\vec{y}^*$.

5.3 Discussion

EG-PQE is similar to the QE algorithm presented in [11]. We will refer to it as *CAV02-QE*. Given a formula $\exists X[F(X,Y)]$, *CAV02-QE* enumerates full assignments to Y. In subspace $\vec{y}$, *CAV02-QE* either adds to F a clause falsified by $\vec{y}$ (if $F_{\vec{y}}$ is unsatisfiable) or generates a plugging clause. To apply the idea of *CAV02-QE* to PQE, we reformulated it in terms of redundancy based reasoning.

The main flaw of *EG-PQE* inherited from *CAV02-QE* is the necessity to use plugging clauses produced from a satisfying assignment. Consider the PQE problem of taking a clause C out of $\exists X[F(X,Y)]$. If F is proved *unsatisfiable* in subspace $\vec{y}$, typically, only a small subset of clauses of $F_{\vec{y}}$ is involved in the proof. Then the clause generated by *EG-PQE* is short and thus proves C redundant in many subspaces different from $\vec{y}$. On the contrary, to prove F *satisfiable* in subspace $\vec{y}$, every clause of $F_{\vec{y}}$ must be satisfied. So, the plugging clause built off a satisfying assignment includes almost every variable of Y. Despite this flaw of *EG-PQE* we present it for two reasons. First, it is a very simple SAT-based algorithm that can be easily implemented. Second, *EG-PQE* has a powerful advantage over *CAV02-QE* since it solves PQE rather than QE. Namely, *EG-PQE* does not need to examine the subspaces $\vec{y}$ where C is implied by $F \setminus \{C\}$. Surprisingly, for many formulas this allows *EG-PQE* to *completely avoid* examining subspaces where F is satisfiable. In this case, *EG-PQE* is very efficient and can solve very large problems. Note that when *CAV02-QE* performs complete QE on $\exists X[F]$, it *cannot* avoid subspaces $\vec{y}$ where $F_{\vec{y}}$ is satisfiable *unless* F itself is unsatisfiable (which is very rare in practical applications).

6 Introducing *EG-PQE*$^+$

In this section, we describe *EG-PQE*$^+$, an improved version of *EG-PQE*.

6.1 Main idea

```
EG-PQE+(F, X, Y, C) {
1   Plg := ∅; F_ini := F
2   while (true) {
.....
11* D := PrvClsRed(y⃗, F, C)
12  Plg := Plg ∪ {D}}}
```

Fig. 2: Pseudocode of *EG-PQE*$^+$

The pseudocode of *EG-PQE*$^+$ is shown in Fig 2. It is different from that of *EG-PQE* only in line 11 marked with an asterisk. The motivation for this change is as follows. Line 11 describes proving redundancy of C for the case when $C_{\vec{y}}$ is not implied by $(F \setminus \{C\})_{\vec{y}}$ and $F_{\vec{y}}$ is satisfiable. Then *EG-PQE* simply uses a satisfying assignment as a proof of redundancy of C in subspace $\vec{y}$. This proof is unnecessarily strong because it proves that *every* clause of F (including C) is redundant in $\exists X[F]$ in subspace $\vec{y}$. Such a strong proof is hard to generalize to other subspaces.

The idea of *EG-PQE*$^+$ is to generate a proof for a much weaker proposition namely a proof of redundancy of C (and only C). Intuitively, such a proof should be easier to generalize. So, *EG-PQE*$^+$ calls a procedure *PrvClsRed* generating

such a proof. *EG-PQE*$^+$ is a generic algorithm in the sense that *any* suitable procedure can be employed as *PrvClsRed*. In our current implementation, the procedure *DS-PQE* [1] is used as *PrvClsRed*. *DS-PQE* generates a proof stating that C is redundant in $\exists X[F]$ in subspace $\vec{y}^* \subseteq \vec{y}$. Then the plugging clause D falsified by $\vec{y}^*$ is generated. Importantly, $\vec{y}^*$ can be much shorter than $\vec{y}$. Appendix F gives a brief description of *DS-PQE*.

Example 3. Consider the example solved in Subsection 5.1. That is, we consider taking clause C_1 out of $\exists X[F(X,Y)]$ where $F = C_1 \wedge \cdots \wedge C_4$, $C_1 = \overline{x}_3 \vee x_4$, $C_2 = y_1 \vee x_3$, $C_3 = y_1 \vee \overline{x}_4$, $C_4 = y_2 \vee x_4$ and $Y = \{y_1, y_2\}$ and $X = \{x_3, x_4\}$. Consider the step where *EG-PQE* proves redundancy of C_1 in subspace $\vec{y} = (y_1 = 1, y_2 = 1)$. *EG-PQE* shows that $(y_1 = 1, y_2 = 1, x_3 = 0)$ satisfies F, thus proving every clause of F (including C_1) redundant in $\exists X[F]$ in subspace $\vec{y}$. Then *EG-PQE* generates the plugging clause $D = \overline{y}_1 \vee \overline{y}_2$ falsified by $\vec{y}$.

In contrast to *EG-PQE*, *EG-PQE*$^+$ calls *PrvClsRed* to produce a proof of redundancy for the clause C_1 alone. Note that F has no clauses resolvable with C_1 on x_3 in subspace $\vec{y}^* = (y_1 = 1)$. (The clause C_2 containing x_3 is satisfied by $\vec{y}^*$.) This means that C_1 is blocked in subspace $\vec{y}^*$ and hence redundant there (see Proposition 2). Since $\vec{y}^* \subset \vec{y}$, *EG-PQE*$^+$ produces a more general proof of redundancy than *EG-PQE*. To avoid re-examining subspace $\vec{y}^*$, *EG-PQE*$^+$ generates a *shorter* plugging clause $D = \overline{y}_1$.

6.2 Discussion

Consider the PQE problem of taking a clause C out of $\exists X[F(X,Y)]$. There are two features of PQE that make it easier than QE. The first feature is that one can ignore the subspaces $\vec{y}$ where $F \setminus \{C\}$ implies C. The second feature is that when $F_{\vec{y}}$ is satisfiable, one only needs to prove redundancy of the clause C alone. Among the three algorithms we run in experiments, namely, *DS-PQE*, *EG-PQE*, and *EG-PQE*$^+$ only *EG-PQE*$^+$ exploits both features. (In addition to using *DS-PQE* inside *EG-PQE*$^+$ we also ran it as a stand-alone PQE solver.) *DS-PQE* does not use the first feature [1] and *EG-PQE* does not exploit the second one. As we show in Sections 8 and 9, this affects the performance of *DS-PQE* and *EG-PQE*.

7 Some Remarks About *START* And Experiments

In [8], we introduced a PQE solver called *START*. One can view *START* as a version of *EG-PQE*$^+$ where the internal procedure *PrvClsRed* is implemented using the machinery of certificate clauses. (Such *PrvClsRed* procedure is more powerful than the one implemented by *DS-PQE* that we use in this paper.) As we mentioned above, we present *EG-PQE* because it is a very simple algorithm that still can efficiently solve some large problems. The reason for introducing *EG-PQE*$^+$ is twofold. First, it helps to emphasize the two advantages of PQE over QE listed in Subsection 6.2. Second, *EG-PQE*$^+$ is a generic algorithm allowing to get new PQE solvers by varying the implementation of *PrvClsRed*.

In the following three sections, we describe experiments with *DS-PQE* (used as a stand-alone PQE algorithm), *EG-PQE* and *EG-PQE*$^+$. The first two sections reproduce the experiments with FIFO buffers and HWMCC-13 benchmarks that we conducted in [8] with *START*. Comparing the results of *DS-PQE*, *EG-PQE* and *EG-PQE*$^+$ allows to better understand which of the features mentioned in Subsection 6.2 makes PQE solving more efficient. We implemented local calls to *DS-PQE* in *EG-PQE*$^+$ using the source of *DS-PQE* provided at [12]. The same source was used to build a binary of *DS-PQE*. The sources of *EG-PQE* and *EG-PQE*$^+$ are available at [13,14]. We used Minisat2.0 [15] as an internal SAT-solver.

8 Experiment With FIFO Buffers[3]

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

Fig. 3: 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. Our objective here is twofold. First, we want to substantiate the intuition of Subsection 3.3 that property generation by PQE (in our case, invariant generation by PQE) has the same reasons to be effective as testing. In particular, by taking out different clauses one generates invariants relating to different parts of the design. So, taking out a clause of the buggy part is likely to produce an unwanted invariant. Second, we want to give an example of an invariant that can be easily identified as unwanted.

8.1 Buffer description

Consider a FIFO buffer that we will refer to as *Fifo*. Let n be the number of elements of *Fifo* and *Data* denote the data buffer of *Fifo*. Let each $Data[i], i = 1, \ldots, n$ have p bits and be an integer where $0 \leq Data[i] < 2^p$. A fragment of the Verilog code describing *Fifo* is shown in Fig 3. 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* has 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 $0 < Val < 2^p$. We assume that the buffer elements are initialized to 0.) So, *Fifo* has an *op-state reachability bug* since it cannot reach operative 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.

[3] All experiments were run on a computer with Intel Core i5-8265U CPU of 1.6 GHz.

8.2 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 *DS-PQE*, *EG-PQE* and *EG-PQE*$^+$ to generate invariants of N as described in Section 4. 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 an op-state reachability bug since every state of *Data* is supposed to be reachable. To generate invariants, we used the formula $F_k = I(S_0) \wedge T(S_0, V_0, S_1) \wedge \cdots \wedge T(S_{k-1}, V_{k-1}, S_k)$ introduced in Subsection 4.2. Here I and T describe the initial state and the transition relation of N respectively and S_j and V_j denote state variables and combinational input variables of j-th time frame respectively. First, we used a PQE solver to generate a local invariant $H(S_k)$ obtained by taking a clause C out of $\exists X_k[F_k]$ where $X_k = S_0 \cup V_0 \cup \cdots \cup S_{k-1} \cup V_{k-1}$. So, $\exists X_k[F_k] \equiv H \wedge \exists X_k[F_k \setminus \{C\}]$. (Since $F_k \Rightarrow H$, no state falsifying H is reachable in k transitions.) In the experiment, we took out only clauses of F_k containing an *unquantified variable* i.e. a variable of S_k. Such a choice was limited but still guaranteed that we **cover** the entire design in terms of **state variables**. 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 generated by PQE, we checked if Q was a global invariant. Namely, we used a public version of *IC3* [16,17] to verify if the property Q held (by showing that no reachable state of N falsified Q). 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. When picking a clause to take out, i.e. a clause with a variable of S_k, 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 and then computed the average value. So, the columns four to twelve of Table 1 actually give the average value of 10 runs.

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

buff. size n	lat-ches	time fra-mes	total *pqe* probs			finished *pqe* probs			unwant. invar			runtime (s.)		
			ds-pqe	*eg-pqe*	*eg-pqe*$^+$	*ds-pqe*	*eg-pqe*	*eg-pqe*$^+$	*ds-pqe*	*eg-pqe*	*eg-pqe*$^+$	*ds-pqe*	*eg-pqe*	*eg-pqe*$^+$
8	300	5	1,236	311	**8**	2%	**37%**	33%	no	**yes**	**yes**	12,141	2,138	**52**
8	300	10	560	737	**39**	2%	1%	**4%**	**yes**	**yes**	**yes**	5,551	7,681	**380**
16	560	5	2,288	2,288	**17**	1%	66%	**69%**	no	no	**yes**	22,612	9,506	**57**
16	560	10	653	2,288	**24**	1%	36%	**39%**	**yes**	no	**yes**	6,541	16,554	**152**

Let us use the first line of Table 1 to explain its structure. The first two columns show the number of elements in *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 next three columns show the total number of PQE problems solved by a PQE solver before an unwanted invariant was generated e.g. 8 problems for *EG-PQE*$^+$. On the other hand, *DS-PQE* failed to find an unwanted invariant and had to solve *all* 1,236 PQE problems of taking out a clause of F_k with an unquantified variable. The following three columns show the share of PQE

problems *finished* in the time limit of 10 sec. For instance, *EG-PQE* finished 37% of 311 problems. The next three columns show if an unwanted invariant was generated by a PQE solver. *DS-PQE* did not find one for the first instance of *Fifo* whereas *EG-PQE* and *EG-PQE*$^+$ found it. The last three columns give the total run time. Table 1 shows that only *EG-PQE*$^+$ managed to generate an unwanted invariant for all four instances of *Fifo*. This invariant asserted that *Fifo* cannot reach a state where an element of *Data* equals *Val*.

9 Experiments With HWMCC Benchmarks

In this section, we describe three experiments with 98 multi-property benchmarks of the HWMCC-13 set [18]. (We use the HWMCC-13 set because it has a multi-property track, see the explanation below.) The number of latches in those benchmarks range from 111 to 8,000. More details about the choice of benchmarks and the experiments can be found in Appendix G. Each benchmark consists of a sequential circuit N and invariants $P_0, \ldots, P_m$ to prove true. Like in Section 4, we call $P_{agg} = P_0 \wedge \cdots \wedge P_m$ the *aggregate invariant*. In experiments 2 and 3 we used PQE to generate new invariants of N. Since every invariant P implied by P_{agg} is a desired one, the necessary condition for P to be *unwanted* is $P_{agg} \not\Rightarrow P$. The conjunction of many invariants P_i produces a stronger invariant P_{agg}, which makes it *harder* to generate P not implied by P_{agg}. (This is the *reason* for using multi-property benchmarks in our experiments.) The circuits of the HWMCC-13 set are *anonymous*. So, we just generated invariants not implied by P_{agg} without deciding if some of them were unwanted.

Similarly to the experiment of Section 8, we used the formula $F_k = I(S_0) \wedge T(S_0, V_0, S_1) \wedge \cdots \wedge T(S_{k-1}, V_{k-1}, S_k)$ to generate invariants. The number k of time frames was in the range of $2 \leq k \leq 10$. As in the experiment of Section 8, we took out only clauses containing a state variable of the k-th time frame. In all experiments, the **time limit** for solving a PQE problem was set to 10 sec.

9.1 Experiment 1

In the first experiment, we generated a formula $H(S_k)$ by taking out a clause C of $\exists X_k[F_k]$ where $X_k = S_0 \cup V_0 \cup \cdots \cup S_{k-1} \cup V_{k-1}$. (So, only the variables of S_k are unquantified in $\exists X_k[F_k]$.) H is a *local invariant* asserting that no state falsifying H can be reached in k transitions. Our goal was to show that PQE can find H for large formulas F_k that have hundreds of thousands of clauses.

We used *EG-PQE* to partition the PQE problems we tried into two groups. *The first group* consisted of 3,736 problems for which we ran *EG-PQE* for 10 sec. and it never encountered a subspace $\vec{s_k}$ where F_k was satisfiable. Here $\vec{s_k}$ is a full assignment to S_k. So, in every subspace $\vec{s_k}$, formula F_k was either unsatisfiable or $(F_k \setminus \{C\}) \Rightarrow C$. (The fact that so many problems meet the condition of the first group came as a big surprise.) *The second group* consisted of 3,094 problems where *EG-PQE* encountered subspaces where F_k was satisfiable. For the first group, *DS-PQE* finished only 30% of the problems within

10 sec. whereas *EG-PQE* and *EG-PQE*$^+$ finished 88% and 89% respectively. The poor performance of *DS-PQE* is due to not checking if $(F_k \setminus \{C\}) \Rightarrow C$ in the current subspace. For the second group, *DS-PQE*, *EG-PQE* and *EG-PQE*$^+$ finished 15%, 2% and 27% of the problems respectively within 10 sec. *EG-PQE* finished far fewer problems because it used a satisfying assignment as a proof of redundancy of C (see Subsection 6.2).

To contrast PQE and QE, we used a high-quality tool *CADET* [19,20] to perform QE on the 98 formulas $\exists X_k[F_k]$ (one formula per benchmark). That is, instead of taking a clause out of $\exists X_k[F_k]$ by PQE, we applied *CADET* to perform full QE on this formula. (Performing QE on $\exists X_k[F_k]$ produces a formula $H(S_k)$ specifying *all* states unreachable in k transitions.) *CADET* finished only 25% of the 98 QE problems with the time limit of 600 sec. On the other hand, *EG-PQE*$^+$ finished 61% of the 6,830 problems of both groups (generated off $\exists X_k[F_k]$) within 10 sec. So, PQE can be much easier than QE if only a small part of the formula gets unquantified.

9.2 Experiment 2

Table 2: Results of invariant generation

pqe solver	#bench marks	results		
		local invar.	glob. invar.	not imp. by P_{agg}
ds-pqe	98	5,556	2,678	2,309
eg-pqe	98	**9,498**	**4,839**	**4,009**
eg-pqe$^+$	98	9,303	4,770	3,937

The second experiment was an extension of the first one. Its goal was to show that PQE can generate invariants for realistic designs. For each clause Q of a local invariant H generated by PQE we used *IC3* to verify if Q was a global invariant. If so, we checked if $P_{agg} \not\Rightarrow Q$ held. To make the experiment less time consuming, in addition to the time limit of 10 sec. per PQE problem we imposed a few more constraints. The PQE problem of taking a clause out of $\exists X_k[F_k]$ terminated as soon as H accumulated 5 clauses or more. Besides, processing a benchmark terminated when the summary number of clauses of formulas H reached 100 or the total run time of all PQE problems generated off $\exists X_k[F_k]$ exceeded 2,000 sec.

Table 2 shows the results of the experiment. The third column gives the number of local single-clause invariants (i.e. the total number of clauses in all H over all benchmarks). The fourth column shows how many local single-clause invariants turned out to be global. (Since global invariants were extracted from H and the summary size of all H could not exceed 100, the number of global invariants per benchmark could not exceed 100.) The last column gives the number of global invariants not implied by P_{agg}. So, these invariants are candidates for checking if they are unwanted. Table 2 shows that *EG-PQE* and *EG-PQE*$^+$ performed much better than *DS-PQE*.

9.3 Experiment 3

To prove an invariant P true, *IC3* conjoins it with clauses $Q_1, \ldots, Q_n$ to make $P \wedge Q_1 \wedge \cdots \wedge Q_n$ inductive. If *IC3* succeeds, every Q_i is an invariant. Moreover,

Q_i may be an *unwanted* invariant. The goal of the third experiment was to demonstrate that PQE and *IC3*, in general, produce different invariant clauses. The intuition here is that *IC3* generates clauses Q_i to prove a *predefined* invariant rather than find an unwanted one. In our experiment, we used *IC3* to generate P^*_{agg}, an *inductive* version of P_{agg}. The experiment showed that in 89% cases, an invariant clause generated by *EG-PQE*$^+$ and not implied by P_{agg} was not implied by P^*_{agg} either. (See Appendix G.3 for more detail.)

10 Properties Mimicking Symbolic Simulation

Let $M(X,V,W)$ be a combinational circuit where X,V,W are internal, input and output variables. In this section, we describe generation of properties of M that mimic symbolic simulation [21]. Every such a property $Q(V)$ specifies a cube of tests that produce the same values for a given subset of variables of W. We chose generation of such properties because deciding if Q is an unwanted property is, in general, simple. The procedure for generation of these properties is slightly different from the one presented in Section 3.

Let $F(X,V,W)$ be a formula specifying M. Let $B(W)$ be a clause. Let $H(V)$ be a solution to the PQE problem of taking a clause $C \in F$ out of $\exists X \exists W[F \wedge B]$. That is $\exists X \exists W[F \wedge B] \equiv H \wedge \exists X \exists W[(F \setminus \{C\}) \wedge B]$. Let $Q(V)$ be a clause of H. Then M has the **property** that for every full assignment $\vec{v}$ to V falsifying Q, it produces an output $\vec{w}$ falsifying B (see Proposition 3 of Appendix B). Suppose, for instance, $Q = v_1 \vee \overline{v}_{10} \vee v_{30}$ and $B = w_2 \vee \overline{w}_{40}$. Then for every $\vec{v}$ where $v_1{=}0, v_{10}{=}1, v_{30}{=}0$, the circuit M produces an output where $w_2 = 0, w_{40} = 1$. Note that Q is implied by $F \wedge B$ rather than F. So, it is a property of M under constraint B rather than M alone.

To generate combinational circuits, we unfolded sequential circuits of the set of 98 benchmarks used in Section 9 for invariant generation. Let N be a sequential circuit. (We reuse the notation of Section 4). Let $M_k(S_0, V_0, \ldots, S_{k-1}, V_{k-1}, S_k)$ denote the combinational circuit obtained by unfolding N for k time frames. Here S_i, V_i are state and input variables of i-th time frame respectively. Let F_k denote the formula $I(S_0) \wedge T(S_0, V_0, S_1) \wedge \cdots \wedge T(S_{k-1}, V_{k-1}, S_k)$ describing the unfolding of N for k time frames. Note that F_k specifies the circuit M_k above under the input constraint $I(S_0)$. Let $B(S_k)$ be a clause. Let $H(S_0, V_0, \ldots, V_{k-1})$ be a solution to the PQE problem of taking a clause $C \in F_k$ out of formula $\exists S_{1,k}[F_k \wedge B]$. Here $S_{1,k} = S_1 \cup \cdots \cup S_k$. That is $\exists S_{1,k}[F_k \wedge B] \equiv H \wedge \exists S_{1,k}[(F_k \setminus \{C\}) \wedge B]$. Let Q be a clause of H. Then for every assignment $(\vec{s}_{ini}, \vec{v_0}, \ldots, \vec{v}_{k-1})$ falsifying Q, the circuit M_k outputs $\vec{s}_k$ falsifying B. (Here $\vec{s}_{ini}$ is the initial state of N and $\vec{s}_k$ is a state of the last time frame.)

In the experiment, we used *DS-PQE*,*EG-PQE* and *EG-PQE*$^+$ to solve 1,586 PQE problems described above. In Table 3, we give a sample of results by *EG-PQE*$^+$. (More details about this experiment can be found in Appendix H.) Below, we use the first line of Table 3 to explain its structure. The first column gives the benchmark name (6s326). The next column shows that 6s326 has 3,342 latches. The third column gives the number of time frames used to produce a

combinational circuit M_k (here $k = 20$). The next column shows that the clause B introduced above consisted of 15 literals of variables from S_k. (Below we still use the index k assuming that $k = 20$.) The literals of B were generated *randomly*. When picking the length of B we just tried to simulate the situation where one wants to set a particular *subset* of output variables of M_k to specified values. The next two columns give the size of the subcircuit M'_k of M_k that feeds the output variables present in B. When computing a property H we took a clause out of formula $\exists S_{1,k}[F'_k \wedge B]$ where F'_k specifies M'_k instead of formula $\exists S_{1,k}[F_k \wedge B]$ where F_k specifies M_k. (The logic of M_k not feeding a variable of B is irrelevant for computing H.) The first column of the pair gives the number of gates in M'_k (i.e. 348,479). The second column provides the number of input variables feeding M'_k (i.e. 1,774). Here we count only variables of $V_0 \cup \cdots \cup V_{k-1}$ and ignore those of S_0 since the latter are already assigned values specifying the initial state $\vec{s}_{ini}$ of N.

Table 3: Property generation for combinational circuits

name	latches	time frames	size of B	subc. M'_k		results			
				gates	inp. vars	min	max	time (s.)	3-val. sim.
6s326	3,342	20	15	348,479	1,774	27	28	3.5	**no**
6s40m	5,608	20	15	406,474	3,450	27	29	1.2	**no**
6s250	6,185	20	15	556,562	2,456	50	54	0.9	**no**
6s395	463	30	15	36,088	569	24	26	1.2	yes
6s339	1,594	30	15	179,543	3,978	70	71	3.7	**no**
6s292	3,190	30	15	154,014	978	86	89	1.2	**no**
6s143	260	40	15	551,019	16,689	526	530	2.9	yes
6s372	1,124	40	15	295,626	2,766	513	518	1.9	**no**
6s335	1,658	40	15	207,787	2,863	120	124	7.8	**no**
6s391	2,686	40	15	240,825	7,579	340	341	9.9	**no**

The next four columns show the results of taking a clause out of $\exists S_{1,k}[F'_k \wedge B]$. For each PQE problem the time limit was set to 10 sec. Besides, *EG-PQE*$^+$ terminated as soon as 5 clauses of property $H(S_0, V_0, \ldots, V_{k-1})$ were generated. The first three columns out of four describe the minimum and maximum sizes of clauses in H and the run time of *EG-PQE*$^+$. So, it took for *EG-PQE*$^+$ 3.5 sec. to produce a formula H containing clauses of sizes from 27 to 28 variables. A clause Q of H with 27 variables, for instance, specifies 2^{1747} tests falsifying Q that produce the same output of M'_k (falsifying the clause B). Here $1747 = 1774 - 27$ is the number of input variables of M'_k not present in Q. The last column shows that at least one clause Q of H specifies a property that cannot be produced by 3-valued simulation (a version of symbolic simulation [21]). To prove this, one just needs to set the input variables of M'_k present in Q to the values falsifying Q and run 3-valued simulation. (The remaining input variables of M'_k are assigned a don't-care value.) If after 3-valued simulation some output variable of M'_k is assigned a don't-care value, the property specified by Q cannot be produced by 3-valued simulation.

Running *DS-PQE*, *EG-PQE* and *EG-PQE*$^+$ on the 1,586 PQE problems mentioned above showed that a) *EG-PQE* performed poorly producing properties only for 28% of problems; b) *DS-PQE* and *EG-PQE*$^+$ showed much better results by generating properties for 62% and 65% of problems respectively. When *DS-PQE* and *EG-PQE*$^+$ succeeded in producing properties, the latter could not be obtained by 3-valued simulation in 74% and 78% of cases respectively.

11 Some Background

In this section, we discuss some research relevant to PQE and property generation. Information on BDD based QE can be found in [22,23]. SAT based QE is described in [11,24,25,26,27,28,29,30,19]. Our first PQE solver called *DS-PQE* was introduced in [1]. It is based on redundancy based reasoning that was first introduced in [31] in terms of variables and then in [32] in terms of clauses. The main flaw of *DS-PQE* is as follows. Consider taking a clause C out of $\exists X[F]$. Suppose *DS-PQE* proved C redundant in a subspace where F is *satisfiable* and some *quantified* variables are assigned. The problem is that *DS-PQE* cannot simply assume that C is redundant every time it re-enters this subspace [33]. The root of the problem is that redundancy is a *structural* rather than semantic property. That is, redundancy of a clause in a formula ξ (quantified or not) does not imply such redundancy in every formula logically equivalent to ξ.

Since *EG-PQE*$^+$ uses *DS-PQE* as a subroutine, the former has the same learning problem as the latter. In [8], we showed that the learning problem above can be potentially solved using the machinery of certificate clauses. So, in the future, the performance of PQE solving can be drastically improved via enhanced learning in subspaces where F is satisfiable. In [8], we also introduced a PQE solver called *START* based on the machinery of certificate clauses. The relation between *EG-PQE*$^+$ and *START* was explained in Section 7.

We are unaware of research on property generation for combinational circuits. As for invariants, the existing procedures typically generate some "auxiliary" invariants helping to prove a predefined property. For instance, they generate loop invariants [34] or invariants relating internal points of circuits checked for equivalence [35]. Another example of auxiliary invariants are clauses generated by *IC3* to make an invariant P inductive [16]. As we showed in Section 9 (experiment 3), the invariants generated by PQE are different from those produced by *IC3*.

12 Conclusions

We consider Partial Quantifier Elimination (PQE) on propositional CNF formulas with existential quantifiers. In contrast to *complete* quantifier elimination, PQE allows to unquantify a *part* of the formula. In this paper, we present two PQE algorithms: *EG-PQE* and *EG-PQE*$^+$. *EG-PQE* is a simple SAT-based algorithm whereas *EG-PQE*$^+$ is a modification of *EG-PQE* that is more powerful and robust. We show that PQE can be used to generate properties of combinational and sequential circuits. Property generation can be viewed as a generalization of testing. Its goal is to check if a design has an *unwanted* property and thus is buggy. We used PQE to generate an unwanted invariant for a buggy FIFO buffer. We also applied PQE to invariant generation for HWMCC benchmarks. Finally, we used PQE to generate properties of combinational circuits mimicking symbolic simulation. Our experiments show that PQE can efficiently generate properties for realistic designs.

References

1. E. Goldberg and P. Manolios, "Partial quantifier elimination," in *Proc. of HVC-14*. Springer-Verlag, 2014, pp. 148–164.
2. 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.
3. K. McMillan, "Interpolation and sat-based model checking," in *CAV-03*. Springer, 2003, pp. 1–13.
4. E. Goldberg, "Property checking by logic relaxation," Tech. Rep. arXiv:1601.02742 [cs.LO], 2016.
5. 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.
6. E. Goldberg, "Equivalence checking by logic relaxation," in *FMCAD-16*, 2016, pp. 49–56.
7. ——, "Property checking without inductive invariant generation," Tech. Rep. arXiv:1602.05829 [cs.LO], 2016.
8. ——, "Partial quantifier elimination by certificate clauses," Tech. Rep. arXiv:2003.09667 [cs.LO], 2020.
9. O. Kullmann, "New methods for 3-sat decision and worst-case analysis," *Theor. Comput. Sci.*, vol. 223, no. 1-2, pp. 1–72, 1999.
10. 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.
11. K. McMillan, "Applying sat methods in unbounded symbolic model checking," in *Proc. of CAV-02*. Springer-Verlag, 2002, pp. 250–264.
12. The source of *DS-PQE*, http://eigold.tripod.com/software/ds-pqe.tar.gz.
13. The source of *EG-PQE*, http://eigold.tripod.com/software/eg-pqe.1.0.tar.gz.
14. The source of *EG-PQE*$^+$, http://eigold.tripod.com/software/eg-pqe-pl.1.0.tar.gz.
15. N. Eén and N. Sörensson, "An extensible sat-solver," in *SAT*, Santa Margherita Ligure, Italy, 2003, pp. 502–518.
16. A. R. Bradley, "Sat-based model checking without unrolling," in *VMCAI*, 2011, pp. 70–87.
17. An implementation of IC3 by A. Bradley, https://github.com/arbrad/IC3ref.
18. HardWare Model Checking Competition 2013 (HWMCC-13), http://fmv.jku.at/hwmcc13/.
19. M. Rabe, "Incremental determinization for quantifier elimination and functional synthesis," in *CAV*, 2019.
20. CADET, https://github.com/MarkusRabe/cadet.
21. R. Bryant, "Symbolic simulation—techniques and applications," in *DAC-90*, 1990, pp. 517–521.
22. ——, "Graph-based algorithms for Boolean function manipulation," *IEEE Transactions on Computers*, vol. C-35, no. 8, pp. 677–691, August 1986.
23. 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.
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. E. Goldberg and P. Manolios, "Quantifier elimination by dependency sequents," in *FMCAD-12*, 2012, pp. 34–44.
32. ——, "Quantifier elimination via clause redundancy," in *FMCAD-13*, 2013, pp. 85–92.
33. E. Goldberg, "Quantifier elimination with structural learning," Tech. Rep. arXiv: 1810.00160 [cs.LO], 2018.
34. I. Dillig, T. Dillig, B. Li, and K. McMillan, "Inductive invariant generation via abductive inference," vol. 48, 10 2013, pp. 443–456.
35. 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.
36. 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.

Appendix

A PQE And Interpolation

In this appendix, we recall the observation of [4] 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* [2]. 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 $Vars(B) \subset Vars(A \wedge B)$. 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.

B Proofs Of Propositions

Proposition 1. *Let H be a solution to the PQE problem of Definition 9. That is $\exists X[F] \equiv H \wedge \exists X[F \setminus G]$. Then $F \Rightarrow H$ (i.e. F implies H).*

Proof. By conjoining both sides of the equality with H one concludes that $H \wedge \exists X[F] \equiv H \wedge \exists X[F \setminus G]$ and hence $H \wedge \exists X[F] \equiv \exists X[F]$. Then $\exists X[F] \Rightarrow H$ and thus $F \Rightarrow H$.

Proposition 2. *Let a clause C be blocked in a formula $F(X,Y)$ with respect to a variable $x \in X$. Then C is redundant in $\exists X[F]$ i.e. $\exists X[F \setminus \{C\}] \equiv \exists X[F]$.*

Proof. It was shown in [9] that adding a clause $B(X)$ blocked in $G(X)$ to the formula $\exists X[G]$ does not change the value of this formula. This entails that removing a clause $B(X)$ blocked in $G(X)$ does not change the value of $\exists X[G]$ either. So, B is redundant in $\exists X[G]$. Let $\vec{y}$ be a full assignment to Y. Then the clause C is either satisfied by $\vec{y}$ or $C_{\vec{y}}$ is blocked in $F_{\vec{y}}$ with respect to x. (The latter follows from the definition of a blocked clause.) In either case $C_{\vec{y}}$ is redundant in $\exists X[F_{\vec{y}}]$. Since $C_{\vec{y}}$ is redundant in $\exists X[F_{\vec{y}}]$ in every subspace $\vec{y}$, C is redundant in $\exists X[F]$.

Proposition 3. *Let $M(X,V,W)$ be a combinational circuit where X,V,W are the internal, input and output variables. Let $F(X,V,W)$ be a formula specifying M. Let $B(W)$ be a clause. Let $H(V)$ be a formula obtained by taking a clause $C \in F$ out of $\exists X \exists W[F \wedge B]$. That is $\exists X \exists W[F \wedge B] \equiv H \wedge \exists X \exists W[(F \setminus \{C\}) \wedge B]$. Let $Q(V)$ be a clause of H. Then for every full assignment $\vec{v}$ to V falsifying Q, the circuit M outputs an assignment $\vec{w}$ falsifying the clause B.*

Proof. From Proposition 1 it follows that $F \wedge B \Rightarrow H$ and hence $F \wedge B \Rightarrow Q$. This entails that $\overline{Q} \Rightarrow \overline{B} \vee \overline{F}$. Let $\vec{v}$ be a full assignment to V i.e. an input to M. Let $(\vec{x},\vec{v},\vec{w})$ be the execution trace produced by M under the input $\vec{v}$. Here $\vec{x}$,$\vec{w}$ are full assignments to X and W respectively. Suppose, $\vec{v}$ satisfies $\overline{Q}$ (and so falsifies Q). Then $(\vec{x},\vec{v},\vec{w})$ satisfies $\overline{B} \vee \overline{F}$. Since $(\vec{x},\vec{v},\vec{w})$ is an execution trace, it satisfies F and so falsifies $\overline{F}$. This entails that $(\vec{x},\vec{v},\vec{w})$ (and specifically $\vec{w}$) satisfies $\overline{B}$ and hence falsifies B.

C Examples Of Problems That Reduce To PQE

In this section, we give a few examples of how a problem can be reduced to PQE.

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

C.2 Equivalence checking by PQE [6]

$N'(X', V', w')$ and $N''(X'', V'', w'')$ be single-output combinational circuits to check for equivalence. Here X', V' are the sets of internal and input variables and w' is the output variable of N'. (Definition of X'', V'', w'' for N'' is the same.) 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 ignoring similarities between N' and N''.

Let $eq(V', V'')$ specify a formula such that $eq(\vec{v}\,', \vec{v}\,'') = 1$ iff $\vec{v}\,' = \vec{v}\,''$ where $\vec{v}\,', \vec{v}\,''$ are full assignments to V' and V'' respectively. Let formulas $G'(X', V', w')$ and $G''(X'', V'', w'')$ specify N' and N'' respectively. (As usual, we assume that a formula G specifying a circuit N is obtained by Tseitin transformations [10], see Section 3.) Let $h(w', w'')$ be a formula obtained by taking eq out of $\exists Z[eq \wedge G' \wedge G'']$ where $Z = X' \cup V' \cup X'' \cup V''$. That is $\exists Z[eq \wedge G' \wedge G''] \equiv h \wedge \exists Z[G' \wedge G'']$. If $h \Rightarrow (w' \equiv w'')$, then N' and N'' are equivalent. Otherwise, N' and N'' are inequivalent, unless they are identical constants i.e. $w' \equiv w'' \equiv 1$ or $w' \equiv w'' \equiv 0$. It is formally proved in [6] that the more similar N', N'' are (where similarity is defined in the most general sense), the easier taking eq out of $\exists Z[eq \wedge G' \wedge G'']$ becomes.

C.3 Model checking by PQE [7]

use PQE to find the reachability diameter of a sequential circuit 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 $T(S', V, S'')$ denote the transition relation of a sequential circuit N where S', S'' are the present and next state variables and V specifies the (combinational) input variables. Let $I(S)$ specify the initial states of N. For the sake of simplicity, we assume that N can stutter i.e. for every state $\vec{s}$ there exists a full assignment $\vec{v}$ to V such that $T(\vec{s}, \vec{v}, \vec{s}) = 1$, (Then the sets of states reachable in m transitions and *at most* m transitions are identical. If T has no stuttering, it can be easily introduced by adding a variable to V.)

Let $Diam(I, T)$ denote the *reachability diameter* for initial states I and transition relation T. That is every state of the circuit N 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 $\exists X_m[I_0 \wedge I_1 \wedge T_m]$. Here I_0 and I_1 specify the initial states of N in terms of variables of S_0 and S_1 respectively, $X_m = S_0 \cup V_0 \cup \cdots \cup S_{m-1} \cup V_{m-1}$ and $T_m = T(S_0, V_0, S_1) \wedge \cdots \wedge T(S_{m-1}, V_{m-1}, S_m)$. If I_1 is redundant, then $Diam(I, T) < m$.

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 generating 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 [36] respectively. In the special case where H meets the three conditions above for $m = 1$, it is an *inductive invariant*.

D Combining PQE With Clause Splitting

In this appendix, we consider combining PQE with clause splitting mentioned in Subsection 3.2. We show that the corresponding PQE problem of taking out a clause produced by splitting is solved by *EG-PQE* in linear time. We also show that if this clause is not redundant, the solution produced by *EG-PQE* is a single-test property.

Here we reuse the notation of Section 3 but, for the sake of simplicity, consider a single-output combinational circuit. Let $M(X, V, w)$ be such a circuit where X and V specify the internal and input variables respectively and w is the output variable of M. Let $F(X, V, w)$ be a formula specifying the circuit M and C be a clause of F. Consider the case of splitting C on *all* variables of V. That is $C = C_1 \wedge \cdots \wedge C_{p+1}$ where $C_1 = C \vee \overline{l(v_1)}, \ldots, C_p = C \vee \overline{l(v_p)}, C_{p+1} = C \vee l(v_1) \vee \cdots \vee l(v_p)$ and $l(v_i)$ is a literal of v_i and $V = \{v_1, \ldots, v_p\}$. Let F' denote the formula obtained from F by replacing the clause C with $C_1 \wedge \cdots \wedge C_{p+1}$. Denote by $\vec{v}_{spl}$ the input assignment falsifying the literals $l(v_1), \ldots, l(v_p)$ where 'spl' stands for 'splitting'.

Consider applying *EG-PQE* to solve the PQE problem of taking the clause C_{p+1} out of $\exists X[F']$. *EG-PQE* starts with looking for an assignment satisfying $(F' \setminus \{C_{p+1}\}) \wedge \overline{C_{p+1}}$ (to find a subspace where $F' \setminus \{C_{p+1}\}$ does not imply C_{p+1}). Consider the following three cases. The first case is that the formula above is unsatisfiable. Then C_{p+1} is trivially redundant in F' and hence in $\exists X[F']$ and *EG-PQE* terminates.

The second case is that there is an assignment $(\vec{x}, \vec{v}_{spl}, w^*)$ satisfying F' where $\vec{x}$ is a full assignment to X and w^* is the output value taken by M under the input $\vec{v}_{spl}$. (Note that any full assignment to V that is different from $\vec{v}_{spl}$ falsifies $\overline{C_{p+1}}$. So, any assignment satisfying $(F' \setminus \{C_{p+1}\}) \wedge \overline{C_{p+1}}$ has to contain $\vec{v}_{spl}$.) Then formula F' is satisfiable in subspace $(\vec{v}_{spl}, w^*)$ and *EG-PQE* adds the plugging clause $D(V, w)$ that is the longest clause falsified by $(\vec{v}_{spl}, w^*)$. If $(F' \setminus \{C_{p+1}\}) \wedge \overline{C_{p+1}} \wedge D$ is unsatisfiable, then C_{p+1} is redundant in $\exists X[F']$ and *EG-PQE* terminates.

The third case occurs when there is an assignment $(\vec{x}, \vec{v}_{spl}, w^*)$ satisfying $(F' \setminus \{C_{p+1}\}) \wedge \overline{C_{p+1}}$ where w^* is the *negation* of the output value taken by M under input $\vec{v}_{spl}$. In this case, formula F' is unsatisfiable in subspace $(\vec{v}_{spl}, w^*)$. Since, $F' \setminus \{C_{p+1}\}$ is satisfiable in this subspace, C_{p+1} is *not* redundant in $\exists X[F']$. To *make* C_{p+1} redundant in subspace $(\vec{v}_{spl}, w^*)$, *EG-PQE* has to add the clause $B(V, w)$ that is the longest clause falsified by $(\vec{v}_{spl}, w^*)$. The clause B is a solution to the PQE problem at hand i.e. $\exists X[F'] \equiv B \wedge \exists X[F' \setminus \{C_{p+1}\}]$.

The clause B above is implied by F' (and hence F) and so, is a **property** of M. This property specifies the input/output behavior of M under the input $\vec{v}_{spl}$. Namely, to satisfy B when the variables of V are assigned as in $\vec{v}_{spl}$, one has to set the variable w to $\overline{w^*}$. The latter is the output produced by M under the input $\vec{v}_{spl}$. So, the property B specifies the behavior of M under a **single test**. In all three cases above, the SAT problem considered by *EG-PQE* is solved just by initial BCP. (The reason is that the formula at hand contains the unit clauses produced by negating C_{p+1} or those specifying the subspace $(\vec{v}_{spl}, w^*)$.) So, *EG-PQE* solves the PQE problem above in **linear** time.

E Tests Specified By A Property Generated By PQE

In this appendix, we show the relation between tests specified by a property obtained via PQE (see Subsection 3.4) and those detecting stuck-at faults. Here, we reuse the notation of Section 3. Let $M(X, V, W)$ be a combinational circuit where X, V, W are the internal, input and output variables respectively. Let $F(X, V, W)$ be a formula specifying M. Let G be an AND gate of M whose functionality is $x_3 = x_1 \wedge x_2$. That is x_1, x_2 are the input variables of G and x_3 is its output variable. The functionality of G is specified by the formula $C_1 \wedge C_2 \wedge C_3$ where $C_1 = \overline{x}_1 \vee \overline{x}_2 \vee x_3$, $C_2 = x_1 \vee \overline{x}_3$, $C_3 = x_2 \vee \overline{x}_3$ (see Example 2). The clauses C_1, C_2, C_3 are present in formula F. Consider taking C_1 out of $\exists X[F]$. This clause makes G produce the output value 1 when its input values are 1. (If x_1 and x_2 are set to 1, the clause C_1 can be satisfied only by setting x_3 to 1.)

Let $H(V, W)$ be the property obtained by taking out C_1. That is $\exists X[F] \equiv H \wedge \exists X[F \setminus \{C_1\}]$. Let $Q(V, W)$ be a clause of H. As we mentioned earlier, we assume that H does not have redundant clauses i.e. those implied by $F \setminus \{C_1\}$. Then the formula $(F \setminus \{C_1\}) \wedge \overline{Q}$ is satisfiable. Let $(\vec{x}, \vec{v}, \vec{w})$ be an assignment satisfying this formula. Note that this assignment *falsifies* C_1. (Indeed, assume the contrary. Then $(\vec{x}, \vec{v}, \vec{w})$ satisfies F because it already satisfies $F \setminus \{C_1\}$. Since this assignment falsifies Q, we have to conclude that $F \not\Rightarrow Q$ and hence $F \not\Rightarrow H$. So we have a contradiction.)

The fact that $(\vec{x}, \vec{v}, \vec{w})$ falsifies C_1 and satisfies $F \setminus \{C_1\}$ means that one can view this assignment as an execution trace of a faulty version M_{flt} of M. Namely, the output x_3 of gate G is stuck at 0 in M_{flt}. (The clause C_1 is falsified when $x_1 = 1, x_2 = 1, x_3 = 0$ i.e. if the gate G outputs 0 when its input variables are assigned 1.) Let $(\vec{x}^*, \vec{v}, \vec{w}^*)$ be the execution trace of M under the input $\vec{v}$. As we showed in Subsection 3.4, $\vec{w}^*$ is different from $\vec{w}$. So the input $\vec{v}$ exposes a stuck-at fault by making M_{flt} and M produce different outputs.

F Brief Description Of *DS-PQE*

In this appendix, we give a high-level view of *DS-PQE* and explain how it works in *EG-PQE*$^+$ in more detail. A full description of *DS-PQE* can be found in [1]. *DS-PQE* is based on the machinery of D-sequents [32] ('*DS*' in the name

DS-PQE stands for 'D-sequent'). Given a formula $\exists X[F(X,Y)]$ and an assignment $\vec{p}$ to $X \cup Y$, a D-sequent is a record $(\exists X[F], \vec{p}\,) \rightarrow C$ stating that clause C is redundant in $\exists X[F]$ in subspace $\vec{p}$. In *EG-PQE*$^+$, *DS-PQE* is called in subspaces $\vec{y}$ where F is satisfiable. (Here $\vec{y}$ is a full assignment to Y.) *DS-PQE* terminates upon deriving a D-sequent $(\exists X[F], \vec{y}^{\,*}) \rightarrow C$ where $\vec{y}^{\,*} \subseteq \vec{y}$. Such derivation means that C is proved redundant in $\exists X[F]$ in subspace $\vec{y}$. Then the plugging clause D falsified by $\vec{y}^{\,*}$ is generated where $Vars(D) = Vars(\vec{y}^{\,*})$.

DS-PQE derives the D-sequent $(\exists X[F], \vec{y}^{\,*}) \rightarrow C$ above by branching on variables of X. A variable is assigned a value either by a decision or during Boolean Constraint Propagation (BCP). A branch of the search tree goes on until an atomic D-sequent is derived for C. This occurs when proving C in the current subspace becomes trivial. When backtracking, *DS-PQE* merges D-sequents derived in different branches using a resolution like operation called *join*. For instance, the join operation applied to D-sequents $(\exists X[F], \vec{p}\,') \rightarrow C$ where $\vec{p}\,' = (y_1 = 0, x_1 = 0)$ and $(\exists X[F], \vec{p}\,'') \rightarrow C$ where $\vec{p}\,'' = (y_2 = 1, x_1 = 1)$ produces the D-sequent $(\exists X[F], \vec{p}\,) \rightarrow C$ where $\vec{p}$ $=(y_1 = 0, y_2 = 1)$.

DS-PQE has three situations where an atomic D-sequent is generated. First, when C is blocked in the current subspace and hence is redundant there. Then an atomic D-sequent $(\exists X[F], \vec{p}) \rightarrow C$ is generated where $\vec{p}$ consists of assignments that made C blocked in the current subspace. For instance, in Example 3, *DS-PQE* would generate an atomic D-sequent $(\exists X[F], (y_1 = 1)) \rightarrow C$. Second, an atomic D-sequent is generated when C is satisfied by an assignment $w = b$ where $w \in X \cup Y$ and $b \in \{0, 1\}$. (This can be a decision assignment or an assignment derived from a clause during BCP.) Then an atomic D-sequent $(\exists X[F], (w = b)) \rightarrow C$ is built. Third, an atomic D-sequent is generated when a conflict occurs and a conflict clause C_{cnfl} falsified in the current subspace is derived. Adding C_{cnfl} to F makes C redundant in the current subspace. So, an atomic D-sequent $(\exists X[F], \vec{p}) \rightarrow C$ is generated where $\vec{p}$ is the shortest assignment falsifying C_{cnfl}.

G Experiments With HWMCC-13 Benchmarks

In Section 9, we described experiments with multi-property benchmarks of the HWMCC-13 set [18]. In this appendix, we provide some additional information. Each benchmark consists of a sequential circuit N and invariants $P_0, \ldots, P_m$ that are supposed to hold for N. We will refer to the invariant P_{agg} equal to $P_0 \wedge \cdots \wedge P_m$ as the *aggregate invariant*. We applied PQE to the generation of invariants of N that may be *unwanted*. Since every invariant P implied by P_{agg} *must* hold, the necessary condition for P to be unwanted is $P_{agg} \not\Rightarrow P$.

Similarly to the experiment of Section 8, we used the formula $F_k = I(S_0) \wedge T(S_0, V_0, S_1) \wedge \cdots \wedge T(S_{k-1}, V_{k-1}, S_k)$ 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 $|F_2| > 500,000$. We also dropped the smallest benchmarks. So, in the experiments, we used 98 out of the 178 benchmarks of the set.

We describe three experiments. In every experiment, we generated properties $H(S_k)$ by taking out a clause of $\exists X_k[F_k]$ where $X_k = S_0 \cup V_0 \cup \cdots \cup S_{k-1} \cup V_{k-1}$. Property H is a *local invariant* claiming that no state falsifying H can be reached in k transitions. As in the experiment of Section 8, we took out only clauses containing an unquantified variable (i.e a variable of S_k). In all experiments, the **time limit** for solving a PQE problem was set to 10 sec.

G.1 Experiment 1

The objective of the *first experiment* was to demonstrate that *EG-PQE*$^+$ could compute H for realistic designs. We also showed in this experiment that PQE could be much easier than QE (see Section 9) and that *EG-PQE*$^+$ outperforms *DS-PQE* and *EG-PQE*. In this experiment, for each benchmark out of 98 mentioned above we generated PQE problems of taking a clause out of $\exists X_k[F_k]$. Some of them were trivially solved by preprocessing. The latter eliminated the blocked clauses of F_k that were easy to identify and ran BCP launched due to the unit clauses specifying the initial state. In all experiments, we *discarded* problems solved by preprocessing i.e. we considered only **non-trivial** PQE problems.

Table 4: PQE problems of the first group

pqe solver	total probl.	finished	
		num.	perc.
ds-pqe	3,736	1,132	30%
eg-pqe	3,736	3,296	88%
eg-pqe$^+$	3,736	**3,325**	**89%**

Let C be a clause taken out of $\exists X_k[F_k]$. We used *EG-PQE* to partition the PQE problems we tried into two groups. *The first group* consisted of problems for which we ran *EG-PQE* with the time limit of 10 sec. and it never encountered a subspace $\vec{s_k}$ where F_k was satisfiable. Here $\vec{s_k}$ is a full assignment to S_k. (Recall that the variables of S_k are the only unquantified variables of $\exists X_k[F_k]$.) So, in every subspace $\vec{s_k}$ tried by *EG-PQE*, formula F_k was either unsatisfiable or $(F_k \setminus \{C\}) \Rightarrow C$. *The second group* consisted of problems where *EG-PQE* encountered subspaces where F_k was satisfiable. For either group we generated up to 40 problems per benchmark. For some benchmarks, the total number of non-trivial problems generated for the first or second group was under 40. Many PQE problems of either group had hundreds of thousands of variables.

Table 5: PQE problems of the second group

pqe solver	total probl.	finished	
		num.	perc.
ds-pqe	3,094	464	15%
eg-pqe	3,094	74	2%
eg-pqe$^+$	3,094	**827**	**27%**

The results for the first group are shown in Table 4. The first column gives the name of a PQE solver. The second column shows the number of PQE problems in the first group. The last two columns give the number and percentage of problems finished in the time limit of 10 sec. Table 4 shows that *EG-PQE* and *EG-PQE*$^+$ performed quite well finishing a very high percentage of problems. The results of *DS-PQE* are much poorer because it does not check if $(F_k \setminus \{C\}) \Rightarrow C$ in the current subspace i.e. if C is trivially redundant.

The results for the second group are shown in Table 5 that has the same structure as Table 4. In particular, the second column gives the number of PQE problems in the second group. Table 5 shows that *EG-PQE* and *EG-PQE*$^+$

finished 2% and 27% of the problems respectively. So, *EG-PQE*$^+$ significantly outperformed *EG-PQE*. The reason is that *EG-PQE* uses a satisfying assignment as a proof of redundancy of the clause C in subspace $\vec{s}_k$.

G.2 Experiment 2

The second experiment was an extension of the first one. Its goal was to show that PQE can generate invariants for realistic designs. For each clause Q of a local invariant H generated by PQE we used *IC3* to verify if Q was a global invariant. If so, we checked if $P_{agg} \not\Rightarrow Q$ held. To make the experiment less time consuming, in addition to the time limit of 10 sec. per PQE problem, we imposed the following constraints. First, we stopped a PQE-solver 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 (i.e. the number of single clauses taken out of $\exists X_k[F_k]$) like in the first experiment, we imposed the following two constraints. First, we stopped processing a benchmark as soon as the total of 100 free clauses was generated (for all the PQE problems generated for this benchmark). Second, we stopped processing a benchmark even earlier if the summary run time for all PQE problems generated for this benchmark exceeded 2,000 sec.

A sample of 9 benchmarks out of the 98 we used in the experiment with *EG-PQE*$^+$ is shown in Table 6. Let us explain the structure of this table by the benchmark 6s306 (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 (7,986). The number of invariants that should hold for 6s306 is provided in the third column (25). So, the aggregate invariant P_{agg} of 6s306 is the conjunction of those 25 invariants. The fourth column shows that the number k of time frames for 6s306 was set to 2 (since $|F_3| > 500,000$). The value 182 shown in the fifth column is the total number of single clauses taken out of $\exists X_k[F_k]$ i.e. the number of PQE problems (where $k = 2$ for 6s306).

Table 6: A sample of HWMCC-13 benchmarks from the experiment with *EG-PQE*$^+$

name	latches	invar. of P_{agg}	time frames	clauses taken out	single-clause properties				
					gen. props	glob. invar.?			not impl. by P_{agg}
						undec.	no	yes	
6s306	7,986	25	2	182	100	0	94	6	6
6s176	1,566	952	3	31	100	23	11	66	11
6s428	3,790	340	4	28	100	9	5	86	83
6s292	3,190	247	5	24	100	41	0	57	57
6s156	513	32	6	113	100	0	0	100	100
6s275	3,196	673	7	20	100	0	50	50	50
6s325	1,756	301	8	22	100	0	1	99	97
6s391	2,686	387	9	35	100	1	29	70	70
6s282	1,933	20	10	111	100	0	64	36	35

Every free clause Q generated by *EG-PQE*$^+$ when taking a clause out of $\exists X_k[F_k]$ was stored as a local single-clause invariant. The sixth column shows that taking clauses out of the scope of quantifiers was terminated when 100 free clauses (specifying 100 local single-clause invariants) were generated. Each of these 100 local invariants held in k-th time frame. The following three columns show how many of those 100 local invariants were true globally. *IC3* finished every problem

out of 100 in the time limit of 30 sec. So, the number of undecided invariants was 0. The number of invariants *IC3* proved false or true globally was 94 and 6 respectively. The last column gives the number of global invariants *not* implied by P_{agg} i.e. invariants that may be unwanted. For 6s306, this number is 6.

G.3 Experiment 3

To prove an invariant P true, *IC3* conjoins it with clauses $Q_1, \ldots, Q_n$ to make $P \wedge Q_1 \wedge \cdots \wedge Q_n$ 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, this nice feature of *IC3* drastically limits the set of invariant clauses it generates. 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* predefined invariants $P_1, \ldots, P_m$ within a time limit. Namely, for every benchmark we formed the aggregate invariant $P_{agg} = P_1 \wedge \cdots \wedge P_m$ and ran *IC3* to prove P_{agg} true.

Table 7: Invariants of *EG-PQE*$^+$ and *IC3*

name	latches	invariants of P_{agg}	glob. single cls. invars		
			glob. invariants	not impl. by P_{agg}	not impl. by P^*_{agg}
6s135	2,307	340	53	53	27
6s325	1,756	301	99	99	96
ex1	130	33	25	16	16
ex2	212	32	64	64	47
6s106	135	17	100	96	96
6s256	3,141	5	13	13	13
ex3	61	3	4	4	4
ex4	63	3	1	1	1
6s209	5,759	2	95	95	89
6s113	994	1	19	16	16
6s143	260	1	97	86	77
6s170	3,141	1	13	13	13
6s252	170	1	54	41	34
Total				**597**	**529**

We selected the benchmarks that *IC3* solved in less than 1,000 sec. (In addition to dropping the benchmarks not solved in 1,000 sec., we discarded those where P_{agg} failed because some invariants P_i were false). Let $\boldsymbol{P^*_{agg}}$ denote the inductive version of P_{agg} produced by *IC3* when proving P_{agg} true. That is, P^*_{agg} is P_{agg} conjoined with the invariant clauses produced by *IC3*. For each of the selected benchmarks we generated invariants by *EG-PQE*$^+$ 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 *EG-PQE*$^+$ were not implied by P_{agg}. The difference from Experiment 2 was that we also checked which global invariants generated by *EG-PQE*$^+$ were not implied by P^*_{agg}.

The results of the experiment are shown in Table 7. The first three columns of this table are the same as in Table 6. They give the name of a benchmark, the number of latches and the number of invariants $P_1, \ldots, P_m$ to prove. (The actual names of examples *ex1*,..,*ex4* in the HWMCC-13 set are *pdtvsarmultip*, *bobtuintmulti*, *nusmvdme1d3multi*, *nusmvdme2d3multi* respectively.) The next column of Table 7 shows the number of local invariants generated by *EG-PQE*$^+$

that turn out to be global. The last two columns give the number of global invariants that were not implied by P_{agg} and P^*_{agg} respectively. The last row of the table shows that in 529 cases out of 597 the invariants not implied by P_{agg} were not implied by P^*_{agg} either. So, in 89% of cases, the invariant clauses generated by *EG-PQE*$^+$ were *different* from those generated by *IC3* to form P^*_{agg}.

H Experiment With Combinational Circuits

In this appendix, we give more information about the experiment with property generation for combinational circuits described in Section 10. We formed PQE problems as follows. For each benchmark N we picked the number k of time frames to unroll. The value of k ranged from 10 to 40. (For larger circuits we picked a smaller value of k.) Then we unrolled N for k time frames to form a combinational circuit M_k and randomly generated a clause $B(S_k)$ of 15 literals. So, B depended on output variables of M_k. After that, we constructed the subcircuit M'_k of M_k as described in Section 10. That is, M'_k was obtained by removing the logic of M_k that did not feed any output variable present in B.

Let formula F'_k specify the subcircuit M'_k. (Here we reuse the notation of Section 10.) For every benchmark, we generated PQE problems of taking different clauses C out of $\exists S_{1,k}[F'_k]$. That is each PQE problem was to find H such that $\exists S_{1,k}[F'_k] \equiv H \wedge \exists S_{1,k}[F'_k \setminus \{C\}]$. Each clause C to take out was chosen among the clauses of F'_k that contained a variable of S_k (i.e. an output variable of M'_k). In this way, we formed a set of 3,254 PQE problems. 1,668 of these problems were solved by simple formula preprocessing i.e. turned out to be trivial. So, in the experiment we used the remaining 1,586 non-trivial PQE problems.

Table 8: Results of property generation

pqe solver	num. of pqe problems	properties were generated			
		number	%	stronger than 3-val. sim.	
				num.	%
ds-pqe	1,586	983	62	728	74
eg-pqe	1,586	450	28	361	**80**
eg-pqe$^+$	1,586	**1,036**	**65**	**809**	78

The time limit for solving a PQE problem was set to 10 sec. Besides, solving a PQE problem terminated as soon as the size of H reached 5 clauses. The results of the experiment are summarized in Table 8. The second column gives the total number of PQE problems. The next two columns show the number and percentage of problems where H was non-empty i.e. had at least one clause. (Recall, that each clause of H represents a property.) The last two columns give the number and percentage of cases where a clause of H represented a property that was stronger than ones produced by 3-valued simulation, a version of symbolic simulation [21]. Consider, for instance, the last line of the table corresponding to *EG-PQE*$^+$. For 809 out of 1,036 PQE problems where H was not empty, at least one clause of H constituted a property that could not be produced by 3-valued simulation. Table 8 shows that *EG-PQE* had the weakest results generating properties only for 28% of problems whereas *DS-PQE* and *EG-PQE*$^+$ performed much better producing properties for 62% and 65% problems respectively.