Partial Quantifier Elimination and Property Generation

Eugene Goldberg(✉)

Land O Lakes, USA
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 one can view 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, we call this problem *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] can be viewed as a special case of PQE as well [4,5].

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

C. Enea and A. Lal (Eds.): CAV 2023, LNCS 13965, pp. 110–131, 2023.
https://doi.org/10.1007/978-3-031-37703-7_6

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, equivalence checking, model checking can be solved in terms of PQE [1,6–8]. 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 can 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 employ *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.

In [7], we showed the viability of an equivalence checker based on PQE. In particular, we presented instances for which this equivalence checker outperformed ABC [9], a high quality tool. In this paper, we describe and check experimentally one more important application of PQE called property generation. Our motivation here is as follows. Suppose a design implementation *Imp* meets the set of specification properties $P_1, \ldots, P_m$. Typically, this set is incomplete. So, *Imp* can still be buggy even if every $P_i, i = 1, \ldots, m$ holds. Let $P^*_{m+1}, \ldots, P^*_n$ be *desired* properties adding which makes the specification complete. If *Imp* meets the properties $P_1, \ldots, P_m$ but is still buggy, a missed property P^*_i above fails. That is, *Imp* has the *unwanted* property $\overline{P^*_i}$. So, one can detect bugs by generating unspecified properties of *Imp* and checking if there is an unwanted one.

Currently, identification of unwanted properties is mostly done by massive testing. (As we show later, the input/output behavior specified by a single test can be cast as a simple property of *Imp*.) Another technique employed in practice is *guessing* unwanted properties that may hold and formally checking if this is the case. The problem with these techniques is that they can miss an unwanted property. In this paper, we describe property generation by PQE. The benefit of PQE is that it can produce much more complex properties than those corresponding to single tests. So, using PQE one can detect bugs that testing overlooks or cannot find in principle. Importantly, PQE generates properties covering different parts of *Imp*. This makes the search for unwanted properties more systematic and facilitates discovering bugs that can be missed if one simply guesses unwanted properties that may hold.

In this paper, we experimentally study generation of invariants of a sequential circuit N. An invariant of N is unwanted if a state that is supposed to be reachable in N falsifies this invariant and hence is unreachable. Note that finding a formal proof that N has no unwanted invariants is impractical. (It is hard to efficiently prove a large set of states reachable because different states are

reached by different execution traces.) So developing practical methods for finding unwanted invariants if very important. We also study generation of properties mimicking symbolic simulation for a combinational circuit obtained by unrolling a sequential circuit. An unwanted property here exposes a wrong execution trace.

This paper is structured as follows. (Some additional information can be found in the supporting technical report [5].) In Sect. 2, we give basic definitions. Section 3 presents property generation for a combinational circuit. In Sect. 4, we describe invariant generation for a sequential circuit. Sections 5 and 6 present *EG-PQE* and *EG-PQE*$^+$ respectively. In Sect. 7, invariant generation is used to find a bug in a FIFO buffer. Experiments with invariant generation for HWMCC benchmarks are described in Sect. 8. Section 9 presents an experiment with property generation for combinational circuits. In Sect. 10 we give some background. Finally, in Sect. 11, we make conclusions and discuss directions for future research.

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 the **set of clauses** $\{C_1, \ldots, C_k\}$. We assume that **every formula is in CNF**.*

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 $Vars(F) \setminus X$.*

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

Definition 4. *A literal, a clause and a formula 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 $\boldsymbol{C_q \equiv 1}$. Otherwise, $\boldsymbol{C_q}$ is the clause obtained from C by removing all literals falsified by $\vec{q}$. Denote by $\boldsymbol{H_q}$ the formula obtained from H by removing the clauses satisfied by $\vec{q}$ and replacing every clause C unsatisfied by $\vec{q}$ with $C_{\boldsymbol{q}}$.*

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* $G_q = H_q$ *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* ***redundant in*** $\boldsymbol{\exists X[F]}$ *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* $\exists X[F]$.

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* $\boldsymbol{\exists X[F] \equiv H \wedge \exists X[F \setminus G]}$. *(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**).*

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.

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.

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 can be found in [5].

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. Let C be a clause of a formula G and $w \in Vars(C)$. The clause C is said to be* ***blocked*** *[10] 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.,* $\exists X[F \setminus \{C\}] \equiv \exists X[F]$.

3 Property Generation by PQE

Many known problems can be formulated in terms of PQE, thus facilitating the design of new efficient algorithms. In [5], we give a short summary of results on solving SAT, equivalence checking and model checking by PQE presented in [1,6–8]. 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 [11]. 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 High-Level View of Property Generation by PQE

One generates properties by PQE until an unwanted property exposing a bug is produced. (Like in testing, one runs tests until a bug-exposing test is encountered.) 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 the design. This increases the probability of discovering a bug. Fourth, every property generated by PQE specifies a large set of high-quality tests.

In this paper (Sects. 7, 9), we consider cases where identifying an unwanted property is easy. However, in general, such identification is not trivial. A discussion of this topic is beyond the scope of this paper. (An outline of a procedure for deciding if a property is unwanted is given in [5].)

3.2 Property Generation as Generalization of Testing

The behavior of M corresponding to a single test can be cast as a property. 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 $B^{\boldsymbol{v}}$ denote the longest clause falsified by $\vec{v}$, i.e., $Vars(B^{\boldsymbol{v}}) = V$. Let $l(w_i)$ be the literal satisfied by the value of w_i produced by M under input $\vec{v}$. Then the clause $B^{\boldsymbol{v}} \vee l(w_i)$ is satisfied by every assignment satisfying F, i.e., $B^{\boldsymbol{v}} \vee l(w_i)$ 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 computed explicitly 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.)

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.

3.3 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 Sects. 5 and 6). Then one can simply stop computing H as soon as the number of clauses in H exceeds a threshold. Such a formula H is still implied by F and hence 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: $C_1 = C \vee \overline{l(v_1)}, \ldots, C_p = C \vee \overline{l(v_p)}$, $C_{p+1} = C \vee l(v_1) \vee \cdots \vee l(v_p)$, 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 $l(v_1), \ldots, l(v_p)$. (The details can be found in [5].)

3.4 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 sampling 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.

3.5 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 [5].

4 Invariant Generation by PQE

In this section, we extend property generation for combinational circuits to sequential ones. Namely, we generate *invariants*. Note that generation of *desired* auxiliary invariants is routinely used in practice to facilitate verification of a predefined property. The problem we consider here is different in that our goal is to produce an *unwanted* invariant exposing a bug. We picked 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.

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 Sect. 7, 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 $R_{\vec{s}}(S)$ 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 Operative State Unreachability by Invariant Generation

In practice, there are two methods to check reachability of op-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 op-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, the latter has an op-state reachability bug. For instance, one can 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 op-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 intuition in Sect. 7.

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. 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 Sect. 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 Sect. 8 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 different parts of N. From now on, when we say "an invariant" without a qualifier we mean a **global invariant**.

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 $(C_1)_{\boldsymbol{y}}$ is redundant in $\exists X[F_{\boldsymbol{y}}]$ (i.e., if C_1 is redundant in $\exists X[F]$ in subspace $\vec{y}$). Note that if $(F \setminus \{C_1\})_{\boldsymbol{y}}$ *implies* $(C_1)_{\boldsymbol{y}}$, then $(C_1)_{\boldsymbol{y}}$ is trivially redundant in $\exists X[F_{\boldsymbol{y}}]$. 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 $(F \setminus \{C_1\})_{\boldsymbol{y}}$ *does not* imply $(C_1)_{\boldsymbol{y}}$.

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 $F_{\boldsymbol{y}}$ is satisfiable. $F_{\boldsymbol{y}} = (\overline{x}_3 \vee x_4) \wedge x_3 \wedge \overline{x}_4$ and so it is *unsatisfiable*. This means that $(C_1)_{\boldsymbol{y}}$ *is not* redundant in $\exists X[F_{\boldsymbol{y}}]$. (Indeed, $(F \setminus \{C_1\})_{\boldsymbol{y}}$ is satisfiable. So, removing C_1 makes F satisfiable in subspace $\vec{y}$.) *EG-PQE makes* $(C_1)_{\boldsymbol{y}}$ redundant in $\exists X[F_{\boldsymbol{y}}]$ 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 $F_{\boldsymbol{y}}$ 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 $F_{\boldsymbol{y}}$ is satisfiable. So, $(C_1)_{\boldsymbol{y}}$

is *already redundant* in $\exists X[F_y]$. 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 $(F \setminus \{C_1\})_y = x_4$ and $(C_1)_y = \overline{x}_3 \vee x_4$, the formula $(F \setminus \{C_1\}) \wedge \overline{C}_1$ is unsatisfiable in subspace $\vec{y}$. In other words, $(C_1)_y$ is implied by $(F \setminus \{C_1\})_y$ 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*

```
EG-PQE(F, X, Y, C) {
1   Plg := ∅; F_ini := F
2   while (true) {
3     G := F \ {C}
4     y⃗ := Sat1(Plg ∧ G ∧ C̄)
5     if (y⃗ = nil)
6       return(F \ F_ini)
7     (x⃗*, B) := Sat2(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*

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 $\exists X[(F \setminus \{C\})_y]$ does not imply F_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_y is redundant because F_y is satisfiable. Second, $(F \setminus \{C\})_y \wedge \overline{C_y}$ is unsatisfiable. In this case, $(F \setminus \{C\})_y$ implies C_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).

If the satisfying assignment $(\vec{y}, \vec{x})$ above exists, *EG-PQE* checks if the formula F_y is satisfiable (line 7). If not, then the clause C_y *is not* redundant in $\exists X[F_y]$ (because $(F \setminus \{C\})_y$ is satisfiable). So, *EG-PQE makes* C_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_y and collecting the literals of Y removed from clauses of this subset when obtaining F_y from F.

If F_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_y means that every clause of F_y including C_y is redundant in $\exists X[F_y]$. 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 that is falsified by $\vec{y}^*$ and involves only the variables assigned in $\vec{y}^*$.

5.3 Discussion

EG-PQE is similar to the QE algorithm presented at CAV-2002 [12]. 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 $\vec{y}$, if $F_{\vec{y}}$ is unsatisfiable, *CAV02-QE* adds to F a clause falsified by $\vec{y}$. Otherwise, *CAV02-QE* generates a plugging clause D. (In [12], D is called "a blocking clause". This term can be confused with the term "blocked clause" specifying a completely different kind of a clause. So, we use the term "the plugging clause" instead.) 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 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

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 where $C_{\vec{y}}$ 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.

```
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 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 $\vec{y}$. (A brief description of *DS-PQE* in the context of *EG-PQE*$^+$ is given in [5].)

Example 3. Consider the example solved in Subsect. 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 the 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 mentioned earlier 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 the latter exploits both features. (In addition to using *DS-PQE* inside *EG-PQE*$^+$ we also run 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 Sects. 7 and 8, this affects the performance of *DS-PQE* and *EG-PQE*.

7 Experiment with FIFO Buffers

In this and the next two sections we describe some experiments with *DS-PQE*, *EG-PQE* and *EG-PQE*$^+$ (their sources are available at [13,14] and [15] respectively). We used Minisat2.0 [16] as an internal SAT-solver. The experiments were run on a computer with Intel Core i5-8265U CPU of 1.6 GHz.

```
      ...
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 threefold. First, we want to give an example of a bug that can be overlooked by testing and guessing the unwanted properties to check (see Subsect. 7.3). Second, we want to substantiate the intuition of Subsect. 3.4 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. Third, we want to give an example of an invariant that can be easily identified as unwanted[2].

7.1 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 $Data[i], i = 1,\dots,n$ have p bits and be an integer where $0 \le Data[i] < 2^p$. 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*.

[2] Let $P(\hat{S})$ be an invariant for a circuit N depending only on a subset $\hat{S}$ of the state variables S. Identifying P as an unwanted invariant is much easier if $\hat{S}$ is meaningful from the high-level view of the design. Suppose, for instance, that assignments to $\hat{S}$ specify values of a high-level variable v. Then P is unwanted if it claims unreachability of a value of v that is supposed to be reachable. Another simple example is that assignments to $\hat{S}$ specify values of high-level variables v and w that are supposed to be *independent*. Then P is unwanted if it claims that some combinations of values of v and w are unreachable. (This may mean, for instance, that an assignment operator setting the value of v erroneously involves the variable w.)

7.2 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 Sect. 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 Subsect. 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 can be reached in k transitions.) In the experiment, we took out only clauses of F_k containing an *unquantified variable*, i.e., a state variable of the k-th time frame. The time limit for solving the PQE problem of taking out a clause was set to 10 s.

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

buff. size n	latches	time frames	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%	**36%**	35%	no	**yes**	**yes**	12,141	2,138	**52**
8	300	10	560	737	**39**	2%	1%	**3%**	**yes**	**yes**	**yes**	5,551	7,681	**380**
16	560	5	2,288	2,288	**16**	1%	65%	**71%**	no	no	**yes**	22,612	9,506	**50**
16	560	10	653	2,288	**24**	1%	36%	**38%**	**yes**	no	**yes**	6,541	16,554	**153**

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* [17,18] 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 for buffers with 32-bit elements are given in Table 1. When picking a clause to take out, i.e., a clause with a state variable of k-th time frame, one could make a good choice by pure luck. To address this issue, we picked clauses to take out *randomly* and performed 10 different runs of invariant generation and then computed the average value. So, the columns four to twelve of Table 1 actually give the average value of 10 runs.

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. For instance,

EG-PQE$^+$ found such an invariant after solving 8 problems. 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 s. For instance, *EG-PQE* finished 36% of 311 problems. The next three columns show if an unwanted invariant was generated by a PQE solver. (*EG-PQE* and *EG-PQE*$^+$ found one whereas *DS-PQE* did not.) 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*.

7.3 Detection of the Bug by Conventional Methods

The bug above (or its modified version) can be overlooked by conventional methods. Consider, for instance, testing. It is hard to detect this bug by *random* tests because it is exposed only if one tries to add *Val* to *Fifo*. The same applies to testing using the *line coverage* metric [19]. On the other hand, a test set with 100% *branch* coverage [19] will find this bug. (To invoke the *else* branch of the *if* statement marked with '*' in Fig. 3, one must set *dataIn* to *Val*.) However, a slightly modified bug can be missed even by tests with 100% branch coverage [5].

Now consider, manual generation of unwanted properties. It is virtually impossible to guess an unwanted *invariant* of *Fifo* exposing this bug unless one knows exactly what this bug is. However, one can detect this bug by checking a property asserting that the element *dataIn* must appear in the buffer if *Fifo* is ready to accept it. Note that this is a *non-invariant* property involving states of different time frames. The more time frames are used in such a property the more guesswork is required to pick it. Let us consider a modified bug. Suppose *Fifo* does not reject the element *Val*. So, the non-invariant property above holds. However, if *dataIn* == *Val*, then *Fifo* changes the *previous* accepted element if that element was *Val* too. So, *Fifo* cannot have two consecutive elements *Val*. Our method will detect this bug via generating an unwanted invariant falsified by states with consecutive elements *Val*. One can also identify this bug by checking a property involving two consecutive elements of *Fifo*. But picking it requires a lot of guesswork and so the modified bug can be easily overlooked.

8 Experiments with HWMCC Benchmarks

In this section, we describe three experiments with 98 multi-property benchmarks of the HWMCC-13 set [20]. (We use this 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 [5]. Each benchmark consists of a sequential circuit N and invariants $P_0, \ldots, P_m$ to prove. Like in Sect. 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 could not know if an unreachable state is supposed to be reachable. For that reason, we just generated invariants not implied by P_{agg} without deciding if some of them were unwanted.

Similarly to the experiment of Sect. 7, 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 Sect. 7, 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 s.

8.1 Experiment 1

In the first experiment, we generated a *local invariant* H by taking out a clause C of $\exists X_k[F_k]$ where $X_k = S_0 \cup V_0 \cup \cdots \cup S_{k-1} \cup V_{k-1}$. The formula H asserts 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* with the time limit of 10 s 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 only the variables S_k are unquantified in $\exists X_k[F_k]$. 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 s 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 s. *EG-PQE* finished far fewer problems because it used a satisfying assignment as a proof of redundancy of C (see Subsect. 6.2).

To contrast PQE and QE, we employed a high-quality tool *CADET* [21,22] 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 s. On the other hand, *EG-PQE*$^+$ finished 60% of the 6,830 problems of both groups (generated off $\exists X_k[F_k]$) within 10 s. So, PQE can be much easier than QE if only a small part of the formula gets unquantified.

8.2 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 s 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 aborted when the summary number of clauses of all formulas H generated for this benchmark reached 100 or the total run time of all PQE problems generated off $\exists X_k[F_k]$ exceeded 2,000 s.

Table 2. 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	4,773	3,940

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*.

8.3 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 twofold. First, *IC3* generates clauses Q_i to prove a *predefined* invariant rather than find an unwanted one. Second, the closer P to being inductive, the fewer new invariant clauses are generated by *IC3*. Consider the circuit N_{triv} that simply stays in the initial state $\vec{s}_{ini}$ (Sect. 4). Any invariant satisfied by $\vec{s}_{ini}$ is already *inductive* for N_{triv}. So, IC3 will not generate *a single new invariant* clause. On the other hand, if the correct circuit is supposed to leave the initial state, N_{triv} has unwanted invariants that our method will find.

In this experiment, we used *IC3* to generate P^*_{agg}, an *inductive* version of P_{agg}. The experiment showed that in 88% cases, an invariant clause generated by *EG-PQE*$^+$ and not implied by P_{agg} was not implied by P^*_{agg} either. (More details about this experiment can be found in [5].)

9 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 [23]. 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 Sect. 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 (a proof of this fact can be found in [5]). 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. The property Q is **unwanted** if there is an input falsifying Q that *should not* produce an output falsifying B.

To generate combinational circuits, we unfolded sequential circuits of the set of 98 benchmarks used in Sect. 8 for invariant generation. Let N be a sequential circuit. (We reuse the notation of Sect. 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_j, V_j are state and input variables of j-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.)

Table 3. Property generation for combinational circuits

name	latches	time frames	size of B	subc. M'_k gates	subc. M'_k inp vars	results min	results max	results time (s.)	results 3-val sim.
6s326	3,342	20	15	348,479	1,774	27	28	2.9	**no**
6s40m	5,608	20	15	406,474	3,450	27	29	1.1	**no**
6s250	6,185	20	15	556,562	2,456	50	54	0.8	**no**
6s395	463	30	15	36,088	569	24	26	0.7	yes
6s339	1,594	30	15	179,543	3,978	70	71	3.1	**no**
6s292	3,190	30	15	154,014	978	86	89	1.1	**no**
6s143	260	40	15	551,019	16,689	526	530	2.5	yes
6s372	1,124	40	15	295,626	2,766	513	518	1.7	**no**
6s335	1,658	40	15	207,787	2,863	120	124	6.7	**no**
6s391	2,686	40	15	240,825	7,579	340	341	8.9	**no**

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 [5].) 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. (Here and 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.

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 s. 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*$^+$ 2.9 s. 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 [23]). 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 66% 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.

10 Some Background

In this section, we discuss some research relevant to PQE and property generation. Information on BDD based QE can be found in [24,25]. SAT based QE is described in [12,21,26–32]. Our first PQE solver called *DS-PQE* was introduced in [1]. It was based on redundancy based reasoning presented in [33] in terms of variables and in [34] 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 [35]. 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 our current implementation of *EG-PQE*$^+$ uses *DS-PQE* as a subroutine, it has the same learning problem. We showed in [36] that this problem can be addressed by the machinery of certificate clauses. So, the performance of PQE can be drastically improved via enhanced learning in subspaces where F is satisfiable.

We are unaware of research on property generation for combinational circuits. As for invariants, the existing procedures typically generate some auxiliary *desired* invariants to prove a predefined property (whereas our goal is to generate invariants that are *unwanted*). For instance, they generate loop invariants [37] or invariants relating internal points of circuits checked for equivalence [38]. Another example of auxiliary invariants are clauses generated by *IC3* to make an invariant inductive [17]. As we showed in Subsect. 8.3, the invariants produced by PQE are, in general, different from those built by *IC3*.

11 Conclusions and Directions for Future Research

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. We show that PQE can be used to generate properties of combinational and sequential circuits. The goal of property generation is to check if a design has an *unwanted* property and thus is buggy. We used PQE to generate an unwanted invariant for a FIFO buffer exposing a non-trivial bug. 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.

There are at least three directions for future research. The first direction is to improve the performance of PQE solving. As we mentioned in Sect. 10, the most promising idea here is to enhance the power of learning in subspaces where the formula is satisfiable. The second direction is to use the improved PQE solvers to design new, more efficient algorithms for well-known problems like SAT, model checking and equivalence checking. The third direction is to look for new problems that can be solved by PQE.

References

1. Goldberg, E., Manolios, P.: Partial quantifier elimination. In: Yahav, E. (ed.) HVC 2014. LNCS, vol. 8855, pp. 148–164. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-13338-6_12

2. Craig, W.: Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. J. Symbolic Logic **22**(3), 269–285 (1957)
3. McMillan, K.L.: Interpolation and SAT-based model checking. In: Hunt, W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 1–13. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45069-6_1
4. Goldberg, E.: Property checking by logic relaxation, Technical report arXiv:1601.02742 [cs.LO] (2016)
5. Goldberg, E.: Partial quantifier elimination and property generation, Technical report arXiv:2303.13811 [cs.LO] (2023)
6. Goldberg, E., Manolios, P.: Software for quantifier elimination in propositional logic. In: ICMS-2014, Seoul, South Korea, 5–9 August 2014, pp. 291–294 (2014)
7. Goldberg, E.: Equivalence checking by logic relaxation. In: FMCAD-2016, pp. 49–56 (2016)
8. Goldberg, E.: Property checking without inductive invariant generation, Technical report arXiv:1602.05829 [cs.LO] (2016)
9. B. L. Synthesis and V. Group: ABC: a system for sequential synthesis and verification (2017). www.eecs.berkeley.edu/~alanmi/abc
10. Kullmann, O.: New methods for 3-SAT decision and worst-case analysis. Theor. Comput. Sci. **223**(1–2), 1–72 (1999)
11. Tseitin, G.: 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., pp. 115–125 (1970)
12. McMillan, K.L.: Applying SAT methods in unbounded symbolic model checking. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 250–264. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45657-0_19
13. The source of *DS-PQE*. http://eigold.tripod.com/software/ds-pqe.tar.gz
14. The source of *EG-PQE*. http://eigold.tripod.com/software/eg-pqe.1.0.tar.gz
15. The source of *EG-PQE*$^+$. http://eigold.tripod.com/software/eg-pqe-pl.1.0.tar.gz
16. Eén, N., Sörensson, N.: An extensible SAT-solver. In: SAT, Santa Margherita Ligure, Italy, pp. 502–518 (2003)
17. Bradley, A.R.: SAT-based model checking without unrolling. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 70–87. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-18275-4_7
18. An implementation of IC3 by A. Bradley. https://github.com/arbrad/IC3ref
19. Aniche, M.: Effective Software Testing: A Developer's Guide. Manning Publications (2022)
20. HardWare Model Checking Competition (HWMCC-2013) (2013). http://fmv.jku.at/hwmcc13/
21. Rabe, M.N.: Incremental determinization for quantifier elimination and functional synthesis. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11562, pp. 84–94. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25543-5_6
22. CADET. http://github.com/MarkusRabe/cadet
23. Bryant, R.: Symbolic simulation–techniques and applications. In: DAC-1990, pp. 517–521 (1990)
24. Bryant, R.: Graph-based algorithms for Boolean function manipulation. IEEE Trans. Comput. **C-35**(8), 677–691 (1986)
25. Chauhan, P., Clarke, E., Jha, S., Kukula, J., Veith, H., Wang, D.: Using combinatorial optimization methods for quantification scheduling. In: Margaria, T., Melham, T. (eds.) CHARME 2001. LNCS, vol. 2144, pp. 293–309. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-44798-9_24

26. Jin, H., Somenzi, F.: Prime clauses for fast enumeration of satisfying assignments to Boolean circuits. In: DAC 2005, pp. 750–753 (2005)
27. Ganai, M., Gupta, A., Ashar, P.: Efficient SAT-based unbounded symbolic model checking using circuit cofactoring. In: ICCAD-2004, pp. 510–517 (2004)
28. Jiang, J.-H.R.: Quantifier elimination via functional composition. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 383–397. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02658-4_30
29. Brauer, J., King, A., Kriener, J.: Existential quantification as incremental SAT. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 191–207. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_17
30. Klieber, W., Janota, M., Marques-Silva, J., Clarke, E.: Solving QBF with free variables. In: Schulte, C. (ed.) CP 2013. LNCS, vol. 8124, pp. 415–431. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40627-0_33
31. Bjorner, N., Janota, M., Klieber, W.: On conflicts and strategies in QBF. In: LPAR (2015)
32. Bjorner, N., Janota, M.: Playing with quantified satisfaction. In: LPAR (2015)
33. Goldberg, E., Manolios, P.: Quantifier elimination by dependency sequents. In: FMCAD-2012, pp. 34–44 (2012)
34. Goldberg, E., Manolios, P.: Quantifier elimination via clause redundancy. In: FMCAD 2013, pp. 85–92 (2013)
35. Goldberg, E.: Quantifier elimination with structural learning, Technical report arXiv: 1810.00160 [cs.LO] (2018)
36. Goldberg, E.: Partial quantifier elimination by certificate clauses, Technical report arXiv:2003.09667 [cs.LO] (2020)
37. Dillig, I., Dillig, T., Li, B., McMillan, K.: Inductive invariant generation via abductive inference, vol. 48, pp. 443–456, October 2013
38. Baumgartner, J., Mony, H., Case, M., Sawada, J., Yorav, K.: Scalable conditional equivalence checking: an automated invariant-generation based approach. In: Formal Methods in Computer-Aided Design, pp. 120–127 (2009)