This webpage describes **Partial Quantifier Elimination (PQE)**.

* PQE is a *generalization* of **Quantifier Elimination (QE)**

* In PQE, one specifies what part of the formula is taken out of the scope of quantifiers (i.e. "unquantified")

* QE can be viewed as a degenerate case of PQE where the *entire* formula gets unquantified.

* PQE provides a language for describing *incremental* computations. Many known problems (e.g. SAT, equivalence checking, model checking) and new problems (e.g. property generation) can be formulated in terms of PQE. Such formulation is important because PQE can be *dramatically* faster than QE.

Contents. (Click on a blue line to expand it. Some browsers, like Firefox, properly backtrack after following a link. Others do not. Then you can simply download this file and use a PDF reader.)

## Definition Of PQE

We describe PQE in terms of **propositional logic** with **existential quantifiers**. We assume that every formula is in **Conjunctive Normal Form (CNF)**, unless otherwise stated. So, every formula $F$ is assumed to be a conjunction of clauses $C_1 \wedge \ldots \wedge C_k$ (a **clause** $C_i$ being a disjunction of literals). We will also view $F$ as the set of clauses $\{C_1, \ldots, C_k\}$.

**Definition 1.** *Let $F(X, Y)$ be a formula where $X$ and $Y$ are sets of variables. Let $H$ be a non-empty subset of clauses of $F$.* ***Partial Quantifier Elimination (PQE)*** *is to find $H^*(Y)$ such that $\boldsymbol{\exists X[F] \equiv H^* \wedge \exists X[F \setminus H]}$. (So, PQE takes $H$ out of the scope of quantifiers in $\exists X[F]$.) $H^*$ is called a* ***solution*** *to PQE. The case of PQE where $H = F$ is called* ***Quantifier Elimination (QE)****. In this case, the entire formula is taken out of the scope of quantifiers.*

The definition of PQE was introduced in [11].

**Example 1.** *Find $H^*(Y)$ such that $\exists X[F] \equiv H^* \wedge \exists X[F \setminus H]$ where*
$Y = \{y_1\}$, $X = \{x_2, x_3\}$, $F = \{C_1, C_2, C_3\}$, $H = \{C_1\}$,
$C_1 = \overline{x}_2 \vee x_3$, $C_2 = y_1 \vee x_2$, $C_3 = y_1 \vee \overline{x}_3$.

*A solution to this problem is $H^* = \{C_4\}$ where $C_4 = y_1$. That is $\exists X[F] \equiv C_4 \wedge \exists X[F \setminus \{C_1\}]$.*

Solving Example 1 is described in Section 6 of [8].

## Relating Complexity Of PQE And QE

In this section, we use redundancy based reasoning to relate the complexity of PQE and QE.

**Definition 2.** *Given a formula $\exists X[F(X,Y)]$, a clause $C$ containing only variables of $Y$ is called a* ***free clause****. If $C$ contains at least one variable of $X$ it is called a* ***quantified clause****.*

Let $\exists X[F(X,Y)]$ be a formula and $H$ be a non-empty subset of quantified clauses of $F$. Let $H^*(Y)$ be a solution to the PQE problem of taking $H$ out of $\exists X[F]$. That is $\exists X[F] \equiv H^* \wedge \exists X[F \setminus H]$. It is not hard to show, that $F$ implies $H^*$. Then $\exists X[F] \equiv H^* \wedge \exists X[F] \equiv H^* \wedge \exists X[F \setminus H]$. That is, a set of free clauses $H^*$ is a solution, if adding $H^*$ makes the clauses of $H$ **redundant** in $\exists X[F]$. This observation is the foundation for *redundancy based reasoning* used in our PQE algorithms described later.

Redundancy based reasoning provides a simple explanation of why PQE is easier than QE. The smaller set $H$, the fewer free clauses need to be added to make $H$ redundant. QE is the hardest case of PQE where $H = F$ and so one needs to take *all clauses* out of the scope of quantifiers in $\exists X[F]$. Decreasing the size of $H$ is only one way to reduce the complexity of PQE. Suppose $H$ already consists of only one clause. Then one can reduce the complexity of PQE even further by clause splitting.

Let $C$ be replaced in $F$ with $k+1$ clauses $C_1, \ldots, C_{k+1}$ where $C_1 = C \vee \overline{v}_1, \ldots$, $C_k = C \vee \overline{v}_k$, $C_{k+1} = C \vee v_1 \vee \ldots \vee v_k$ and $v_i, i = 1, \ldots, k$ are variables of $F$. (Obviously, $C \equiv C_1 \vee \ldots \vee C_{k+1}$.) The idea here is to take out clause $C_{k+1}$ instead of the original clause $C$. Since $C_{k+1}$ contains many more literals than $C$, making the former redundant can be dramatically simpler than the latter. In [7], we show that clause splitting can reduce the complexity of PQE to *linear*.

## Applications Of PQE

In this section, we justify our interest in PQE by giving a list of problems that can be reduced to PQE. (This list is far from being complete.)

- SAT
- equivalence checking
- model checking
- property generation (a general idea)
- generation of safety properties

## SAT-solving by PQE

In this subsection, we describe reduction of SAT to PQE introduced in [11]. Consider the SAT problem of checking if formula $\exists X[F(X)]$ is satisfied. Note that SAT is a special case of QE where all the variables are quantified. In terms of redundancy based reasoning, in SAT, one needs to prove *all clauses* of $F$ redundant in $\exists X[F]$. This is done either via derivation and adding an empty clause ($F$ is unsatisfiable) or without such derivation ($F$ is satisfiable). So, from a purely formal point of view, reduction of SAT to PQE can be very beneficial if only a *small subset* of clauses of $F$ needs to be proved redundant in $\exists X[F]$.

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

Taking out the formula $H$ above is a rather naive way to solve SAT. Fortunately, it can be improved in many directions. Consider, for instance, the case when $F$ is satisfiable. Then one needs to prove that $\exists X[F] \equiv \exists X[F \setminus H]$ i.e. show that $H$ can simply be dropped from $\exists X[F]$. In reality, it suffices to show that $H$ can be dropped in *some* subspace $\vec{q}$ where the assigned variables of $X$ have the same value as in $\vec{x}$. (If $H$ can be dropped in the subspace $\vec{q}$, then $F$ is satisfiable there.)

## Equivalence checking by PQE

In this subsection, we recall a procedure for equivalence checking based on PQE [4]. The appeal of this procedure is that it is *complete* and yet can fully exploit the *structural similarity* of circuits to be checked for equivalence. The existing equivalence checking algorithms exploiting such similarity(e.g. [13]) are *incomplete*. The reason is that they use a very narrow definition of similarity (e.g. the one requiring the existence of functionally equivalent cut points).

Let $N'(X', Y', z')$ and $N''(X'', Y'', z'')$ be single-output combinational circuits to check for equivalence. Here $X', X''$ are sets of input variables, $Y', Y''$ are sets of internal variables and $z', z''$ are output variables of $N'$ and $N''$ respectively. Let $eq(X', X'')$ specify a formula such that $eq(\vec{x}\,', \vec{x}\,'') = 1$ iff $\vec{x}\,' = \vec{x}\,''$ where $\vec{x}\,', \vec{x}\,''$ are full assignments to $X'$ and $X''$. Let formulas $G'(X', Y', z')$ and $G''(X'', Y'', z'')$ specify $N'$ and $N''$ respectively. (As usual, we assume that a formula $G$ specifying a circuit $N$ is obtained by Tseitin transformations [16].)

Let $h(z', z'')$ be a solution to the PQE problem of taking the formula $eq$ out of $\exists W[eq \wedge G' \wedge G'']$ where $W = X' \cup Y' \cup X'' \cup Y''$. That is $\exists W[eq \wedge G' \wedge G''] \equiv h \wedge \exists W[G' \wedge G'']$ . If $h \Rightarrow (z' \equiv z'')$, then $N'$ and $N''$ are equivalent. Otherwise, $N'$ and $N''$ are inequivalent, unless they are identical constants (i.e. $z' \equiv z'' \equiv 1$ or $z' \equiv z'' \equiv 0$). As we showed in [4], the complexity of taking out $eq$ reduces *exponentially* if $N'$ and $N''$ are structurally similar. We also showed experimentally that a PQE based equivalence checker easily solved instances that were hard for a well-known tool called ABC [15].

## Model checking by PQE

This subsection discusses model checking by PQE. Namely, we sketch a technique where a safety property is proved *without* generation of an inductive invariant [6]. This technique can be viewed as a generalization of IC3 [2].

Let formulas $T(S_j, S_{j+1})$ and $I(S_0)$ specify the transition relation and initial states of a transition system $\xi$. Here $S_j$ denotes the set of state variables of $j$-th time frame. For the sake of simplicity, we assume that $\xi$ is able to stutter. Let $Diam(I, T)$ denote the *reachability diameter* for initial states $I$ and transition relation $T$. That is every state of the system $\xi$ can be reached in at most $Diam(I, T)$ transitions. Given a number $m$, **one can use PQE** to decide if $Diam(I, T) \leq m$. This is done by checking if $I_1$ is redundant in $\exists \mathbb{S}_{m-1}[I_0 \wedge I_1 \wedge \mathbb{T}_m]$. Here $I_0$ and $I_1$ are initial states in terms of variables of $S_0$ and $S_1$ respectively, $\mathbb{S}_{m-1} = S_0 \cup \ldots \cup S_{m-1}$ and $\mathbb{T}_m = T(S_0, S_1) \wedge \ldots \wedge T(S_{m-1}, S_m)$. If $I_1$ is redundant, then $Diam(I, T) \leq m$ holds. Note that **no generation of all reachable states** is required here.

One can use the idea above for a model checker. Let $P$ be a safety property to prove and $H$ denote a constrained version of $P$ where $I \Rightarrow H \Rightarrow P$. In **IC3**, one constrains $P$ to $H$ to make the latter an *inductive* invariant. In reality, it suffices to find $H$ such that $I \Rightarrow H$ and no $\overline{P}$-state, i.e. a state falsifying $P$, can be reached from an $H$-state. The existence of such $H$ means that $P$ holds. So, $H$ *does not have to be* an invariant, let alone, an inductive one.

Proving that $H$ meets the requirement at hand is done in two steps. First, one uses PQE to find $m$ such that $Diam(H, T) \leq m$. Then BMC [1] is used to show that no $\overline{P}$-state is reachable from an $H$-state in $m$ transitions or less. Here is some intuition behind this approach.

Stuttering means that $T(\vec{s}, \vec{s}) = 1$, for every state $\vec{s}$. Then the sets of states reachable in $m$ transitions and *at most* $m$ transitions are identical. If $T$ does not have the stuttering feature it can be easily introduced.

Note, that one can simply set $H$ to $I$. However, then $m$ equals the reachability diameter of $\xi$ and so can be very large. If $H$ is an inductive invariant, then $m = 1$, but finding $H$ can be very hard. So, it makes sense to find the golden middle between these two extremes (i.e. to construct a formula $H$ that is not an inductive invariant but for which the value of $m$ is not too large).

## Property generation by PQE (a general idea)

In this subsection, we use the example of combinational design to introduce property generation by PQE [7]. Later we describe how this idea works for generation of safety properties for sequential circuits.

Let $Sp(X, Z)$ be a specification of a combinational circuit where $X$ and $Z$ are sets of input and output variables. We assume that $Sp$ gives some (partial) description of the circuit to design. Let $N(X, Y, Z)$ be an implementation of $Sp$. (Here $Y$ specifies the internal variables of $N$.) Suppose that one formally proves that $N$ meets $Sp$. This does not guarantee that $N$ is correct if $Sp$ is *incomplete*. So, the formal part of verification is followed by mandatory testing. Usually, the latter is guided by some coverage metric.

A straightforward way to improve the *formal* component of verification is to compute the truth table of $N$ to check if it is correct. Let $F(X, Y, Z)$ be a formula specifying the circuit $N$ obtained by Tseitin transformations [16]. That is $F(\vec{x}, \vec{y}, \vec{z}) = 1$ iff $\vec{y}$,$\vec{z}$ specifies the execution trace of $N$ under the input assignment $\vec{x}$. The truth table $F^*(X, Z)$ of $N$ is obtained by performing QE on $\exists Y[F]$. Unfortunately, for large circuits, finding $F^*$ is very hard if not impossible.

One can address the problem above using the following idea. The truth table $F^*$ can be viewed as the strongest property of $N$. Instead, one can try to generate *weaker properties* of $N$ by PQE. (The objective here is to find a "bad" property indicating the presence of a bug.) This can be done as follows. Let formula $Q(X, Z)$ be obtained by taking a clause $C$ out of the scope of quantifiers in $\exists Y[F]$. Namely, $\exists Y[F] \equiv Q \wedge \exists Y[F \setminus \{C\}]$. Since $Q$ is implied by $F$, the former is a **property** of $N$. It is a **bad property** if $Q(\vec{x}, \vec{z}) = 0$ and $N$ is supposed to produce $\vec{z}$ for the input $\vec{x}$. (The existence of property $Q$ implies that $N$ lacks some "good input/output behaviors".) We assume here that the specification $Sp$ does not determine the output of $N$ for the input $\vec{x}$. So, the decision whether $Q$ is a bad property is taken by the designer.

The appeal of PQE here is twofold. First, taking out only one clause of $F$ can be dramatically simpler than QE that takes out *all* clauses. Second, by taking out different clauses one can **cover** the entire design like it is done in testing with a coverage metric. The intuition here is that by taking out a clause belonging to a buggy part of $N$ one can produce a bad property $Q$.

Note that the input/output behavior of $N$ corresponding to a single test can be cast as a property. The latter can be generated by splitting $C \in F$ into a set of clauses $C_1, \ldots, C_{k+1}$ and taking out $C_{k+1}$ instead of $C$. So, by using PQE one can generate properties ranging from the weakest ones (describing the input/output behavior of a single test) to the strongest property (specifying the truth table).

Indeed, let $\vec{x}$ be a test (i.e. a full assignment to $X$). Let $\vec{z}$ be the output produced by $N$ for $\vec{x}$. Let $Q(X, Z)$ be the property such that

- $Q(\vec{x'}, \vec{z'}) = 1$ if $\vec{x'} \neq \vec{x}$.
- $Q(\vec{x'}, \vec{z'}) = 1$ if $\vec{x'} = \vec{x}$ and $\vec{z'} = \vec{z}$
- $Q(\vec{x'}, \vec{z'}) = 0$ if $\vec{x'} = \vec{x}$ and $\vec{z'} \neq \vec{z}$

The property $Q$ essentially states that $N$ outputs $\vec{z}$ for the input $\vec{x}$.

## Generation of safety properties

In the previous subsection, we introduced the idea of property generation by PQE. Here we apply this idea to generation of safety properties for a sequential circuit [8]. So, in this subsection, by properties we mean **safety properties**.

Let $P_1(S), \ldots, P_k(S)$ be a set of properties of a sequential circuit to design where $S$ is the set of state variables. One can view $Sp = P_1 \wedge \ldots \wedge P_k$ as a **specification**. We call $Sp$ a *complete specification* (in terms of reachable states) if $Sp(\vec{s}) = 1$ entails that state $\vec{s}$ must be reachable in an implementation of $Sp$.

Let $N$ be a sequential circuit that meets $Sp$. This essentially means that $N$ does not reach a bad state (i.e. a state falsifying $Sp$). However, this *does not* mean that $N$ reaches all required good states. For instance, $N$ meets $Sp$ by simply staying in an initial state satisfying $Sp$. So, in practice, checking that $N$ does what it is supposed to do is performed **by testing**.

Ideally, one can show that $N$ reaches all required states by computing a formula $R$ defining all reachable states. If $Sp \Rightarrow R$, then $N$ is correct (in terms of reachable states). However, constructing $R$ for a large circuit is hard if not infeasible. Besides, $Sp$ is typically incomplete. In this case, $Sp \Rightarrow R$ may not hold even if $N$ is correct. One can mitigate the problem above by generating properties of $N$ (instead of computing $R$). The existence of a bad property $Q(S)$ means that $N$ is buggy. $Q$ is a **bad property** if there is a state $\vec{s}$ that is supposed to be reachable and $Q(\vec{s}) = 0$. (So, $\vec{s}$ is actually unreachable in $N$.) Note that the unreachability of a state **cannot be detected** by a counterexample. So, finding the underlying bug by testing is hard if not impossible.

If $Sp$ is complete, a property $Q$ is bad iff $Sp \not\Rightarrow Q$. Otherwise, $Sp \not\Rightarrow Q$ is only the necessary condition for $Q$ to be a bad property. (In this case, the *designer* must make a decision if a property $Q$ is bad.) One can build a diverse set of properties relating to different parts of $N$ as described earlier. First, one builds a formula with existential quantifiers defining the functionality of $N$. Then one generates properties of $N$ by taking clauses out of the scope of quantifiers. Here is this process in more detail.

Let $I$ and $T$ denote the initial states and transition relation of $N$. Let formula $F_m$ be obtained by unfolding $N$ for $m$ time frames. That is $F_m = I(S_0) \wedge T(S_0, S_1) \wedge \ldots \wedge T(S_{m-1}, S_m)$ where $S_j$ denotes the state variables of $j$-th time frame, $0 \leq j \leq m$. Let $G_m(S_m)$ be a solution to the PQE problem of taking a clause $C$ out of $\exists \mathbb{S}_{m-1}[F_m]$ where $\mathbb{S}_{m-1} = S_0 \cup \ldots \cup S_{m-1}$. That is $\exists \mathbb{S}_{m-1}[F_m] \equiv G_m \wedge \exists \mathbb{S}_{m-1}[F_m \setminus \{C\}]$. Since $F_m$ implies $G_m$, the latter is a **local property** of $N$ holding in $m$-th time frame. That is a state falsifying $G_m$ is unreachable in $N$ in $m$ transitions.

Note that every clause $Q$ of $G_m$ is itself a local property. Importantly, $Q$ can be an **invariant** (i.e. hold in *every* time frame) even if $G_m$ is not. Our experiments [8] showed that even for small $m$, many clauses of $G_m$ were an invariant. To find out if a property $Q$ holds for $N$ (and so $Q$ is an invariant) one can run a model checker. If $Q$ is an invariant not implied by $Sp$, the designer should decide if $Q$ is a bad property. By taking out different clauses of $F_m$ to build local properties $G_m$ and extracting single-clause invariants $Q$, one generates a diverse set of properties of $N$.

## Contrasting PQE with SAT

- PQE and SAT can be viewed as two different ways to cope with the complexity of QE.
- SAT is a degenerate case of QE where *all variables* are quantified. This makes SAT simpler to solve at the expense of losing the semantic power of QE.
- On the contrary, PQE *generalizes* QE. The latter can be viewed as a degenerate case of PQE where the *entire formula* gets unquantified. So, PQE has more semantic power than QE.
- On the other hand, PQE can be dramatically simpler than QE if only a small part of the formula is unquantified. So, the appeal of PQE is that
  - it can potentially be as efficient as a SAT-solver and
  - it has even more semantic power than QE.

## Interpolation As A Special Case Of PQE

In this section, we recall the observation of [5] that interpolation is a special case of PQE. Let $A(X,Y) \wedge B(Y,Z)$ be an unsatisfiable formula. Let $I(Y)$ be a formula such that $A \wedge B \equiv I \wedge B$ and $A \Rightarrow I$. Then $I$ is called an *interpolant* [3].

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.

- One does not assume that $A \wedge B$ is *unsatisfiable*.
- One does not assume that $A \wedge B$ depends on *more variables* than $B$. That is, in general, the objective of PQE is *not* to eliminate variables.
- A solution $A^*$ is implied by $A \wedge B$ rather than by $A$ *alone*.

So, one can view interpolation as a special case of PQE.

## PQE solvers

In this section, we briefly describe two PQE-solvers called *DS-PQE* and *START*. A Linux binary of *START* can be downloaded from [18]. Consider the problem of taking the set of clauses $H$ out of $\exists X[F(X,Y)]$. That is one needs to find a formula $H^*(Y)$ such that $\exists X[F] \equiv H^* \wedge \exists X[F \setminus H]$. Either solver employs redundancy based reasoning. Namely, it constructs $H^*$ as a set of clauses implied by $F$ that makes $H$ redundant in $\exists X[F]$.

*DS-PQE* was introduced in [11]. It is based on the machinery of D-sequents [9, 10, 12]. A D-sequent is a record $(\exists X[F], \vec{q}) \rightarrow C$ stating that a clause $C \in F$ is redundant in $\exists X[F]$ in subspace $\vec{q}$. For every quantified clause $C$ of $H$, *DS-PQE* derives the D-sequent $(\exists X[F], \emptyset) \rightarrow C$. The latter states redundancy of $C$ in the entire space. *DS-PQE* is a branching algorithm. It assigns variables of $X \cup Y$ until a subspace $\vec{q}$ is reached where every clause $C$ of $H$ is proved redundant. In some cases, proving $H$ redundant requires adding a new clause. (By proving $H$ redundant in a subspace, we mean either showing that $H$ is *already* redundant in this subspace or *making* $H$ redundant by adding a new clause to $F$.) A resolution-like operation called *join* can be applied to merge D-sequents derived in different subspaces. The free clauses one needed to add to $F$ to derive the D-sequents proving redundancy of $H$ in the entire space form a solution $H^*$.

*DS-PQE* has two flaws. First, it backtracks only upon reaching a subspace where *all clauses* of $H$ are proved redundant. This often leads to constructing very large search trees. Second, reusing a D-sequent in a new subspace requires keeping a lot of contextual information. This makes such reusing quite expensive. To address these flaws, we developed a PQE algorithm called *START* (Single TARgeT) [8]. At any given moment, *START* proves redundancy of only one clause (hence the name "single target"). Let $C_{trg}$ denote the current target clause. To certify redundancy of $C_{trg}$ in a subspace $\vec{q}$, *START* derives a clause $K$ that implies $C_{trg}$ in the subspace $\vec{q}$. The clause $K$ is called a **certificate**. Importantly, reusing certificates is much simpler than D-sequents.

*START* is somewhat similar to a SAT-solver. It assigns variables of $X \cup Y$ and runs Boolean Constraint Propagation (BCP). When a backtracking condition is met, *START* does some learning and then backtracks. The main difference here is that the goal of *START* is to prove redundancy of $H$ rather than find a satisfying assignment. Clauses of $H$ are proved redundant one by one (i.e. every quantified clause of $H$ serves as $C_{trg}$ at some point). *START* backtracks as soon as the current target clause $C_{trg}$ is proved redundant. Before backtracking, *START* derives a certificate clause $K$ that implies $C_{trg}$ in the current subspace. (So, $K$ certifies the redundancy of $C_{trg}$.)

*START* has the three backtracking conditions listed below.

- A conflict occurs. In this case, *START* derives a conflict certificate (that is similar to a conflict clause derived by a SAT-solver). This certificate is added to the formula $F$.

- An existing clause of $F$ implies $C_{trg}$ in the current subspace. *START* uses this clause to derive a *non-conflict* certificate $K$. (That is $K$ is not

falsified in the current subspace.)

- $C_{trg}$ is blocked in the current subspace. This means that all clauses of $F$ that can be resolved with $C_{trg}$ on some quantified variable are either satisfied or proved redundant. In this case, $START$ derives a non-conflict certificate.

By resolving local certificates derived in different subspaces, $START$ produces a "global" certificate implying $C_{trg}$ in the entire space. $START$ terminates after deriving a global certificate for every clause of formula $H$. The free clauses one needed to add to $F$ to derive a global certificate for every quantified clause of $H$ form a solution $H^*$.

## Some Experimental Results

In this section, we present results of an experiment described in [8]. In this experiment, we used a PQE solver to generate properties for 112 sequential circuits of the HWMCC-13 multi-property benchmark set. For every circuit $N$, we constructed a formula $F_m$ obtained by unfolding $N$ for $m$ time frames. That is $F_m = I(S_0) \wedge T(S_0, S_1) \wedge \ldots \wedge T(S_{m-1}, S_m)$ where $I$ and $T$ specify initial states and transition relation and $S_j$ denotes the state variables of $j$-th time frame. The value of $m$ was in the range of $2 \leq m \leq 10$. Namely, $m$ was the largest value in this range for which the size of $F_m$ did not exceed 500,000 clauses.

Table 1: A sample of HWMCC-13 benchmarks. 50 PQE problems and 1 QE problem per benchmark. The time limit is 5 sec. for PQE and 600 sec. for QE.

| name | latches | time frames | formula $F_m$ after prepross. | | PQE (start) solved | | QE (cadet) solved? |
|---|---|---|---|---|---|---|---|
| | | | variab. | clauses | yes | no | |
| 6s380 | 5,606 | 2 | 7,366 | 9,907 | 50 | 0 | yes |
| 6s176 | 1,566 | 3 | 15,754 | 39,704 | 49 | 1 | no |
| 6s428 | 3,790 | 4 | 92,274 | 231,506 | 16 | 34 | no |
| 6s292 | 3,190 | 5 | 12,226 | 23,645 | 50 | 0 | no |
| 6s156 | 513 | 6 | 80,944 | 237,235 | 0 | 50 | no |
| 6s275 | 3,196 | 7 | 49,130 | 109,328 | 20 | 30 | no |
| 6s325 | 1,756 | 8 | 102,241 | 257,867 | 2 | 48 | no |
| 6s391 | 2,686 | 9 | 63,265 | 154,154 | 38 | 12 | no |
| 6s372 | 1,124 | 10 | 17,380 | 39,088 | 50 | 0 | no |

For every formula $F_m$, we performed trivial pre-processing. The latter ran BCP to satisfy the unit clauses (introduced to $F_m$ by the formula $I$) and removed blocked clauses. For the resulting formula $F_m$ we generated PQE problems by taking out a clause from $\exists \mathbb{S}_{m-1}[F_m]$ where $\mathbb{S}_{m-1} = S_0 \cup \ldots \cup S_{m-1}$. (The formula $\exists \mathbb{S}_{m-1}[F_m]$ defines the states of $N$ reachable in $m$ transitions.) To make the experiment less time consuming, we generated at most 50 PQE problems per formula $F_m$ (i.e. per benchmark). Besides, we limited the run time of PQE solving to 5 sec per problem. In addition to testing the performance of *START*, we also compared the complexity of PQE and QE. We used *CADET* [14, 17] to perform QE on 112 formulas $\exists \mathbb{S}_{m-1}[F_m]$. That is, instead of taking a clause out of $\exists \mathbb{S}_{m-1}[F_m]$ by PQE, we applied *CADET* to perform full QE on this formula. The time limit for QE was set to 600 sec.

Table 1 shows the results of this experiment for a sample of benchmarks. (The summary of results on all 112 benchmarks is described here.) The first column gives the name of a benchmark. The second column shows the number of latches in this benchmark. The third column provides the value of $m$ in $F_m$ (i.e. the number of time frames). The next two columns give the size of formula $F_m$ in terms of variables and clauses after pre-processing. The following two columns show how many PQE problems (out of 50) were solved by *START* in 5 sec. The last column indicates whether *CADET* was able to finish QE on the formula $F_m$ in 600 sec.

Table 1 shows that *START* was capable to solve PQE problems for large

formulas. Besides, as expected, PQE turned out to be dramatically simpler than QE due to the fact that only one clause was taken out of the scope of quantifiers. *CADET* finished 1 out of 9 QE problems in 600 secs. On the other hand, *START* solved 266 out of 450 PQE problems in 5 secs.

Table 2: Total results. Time limits: 5sec. for PQE and 600 sec. for QE

| solver name | solver type | bench-marks | prob-lems | solved | unsol-ved |
|---|---|---|---|---|---|
| *start* | PQE | 112 | 5,418 | 3,102 | 2,316 |
| *cadet* | QE | 112 | 112 | 32 | 80 |

The results for all 112 benchmarks are shown in Table 2. The first two columns give the name of a solver and its type. The next two columns show the total number of benchmarks and problems. The last two columns provide the number of problems solved and unsolved in the time limit. Table 2 shows that *START* managed to solve 57% of the problems within 5 secs. For 92 benchmarks out of 112, at least one PQE problem generated off $\exists \mathbb{S}_{k-1}[F_k]$ was solved by *START* in the time limit. *CADET* solved only 32 out of 112 QE problems with the time limit of 600 sec. For many formulas $\exists \mathbb{S}_{m-1}[F_m]$ for which *CADET* failed to finish QE in 600 sec, *START* solved all 50 PQE problems generated off $\exists \mathbb{S}_{m-1}[F_m]$ in 5 sec.

## Status Quo

- In our work on PQE we are pursuing two directions. First, we search for problems that can be reduced to PQE. Second, we work on improving the quality of PQE solving.
- We have found numerous problems that can be solved in terms of PQE such as SAT, equivalence checking, model checking, test generation, property generation and so on.
- We developed a PQE algorithm called *START* based on redundancy based reasoning implemented via the machinery of certificate clauses. Currently, *START* is in its infancy. So, its performance can be dramatically improved.
- The strongest results we achieved in applying PQE so far are in equivalence checking and property generation.
- In equivalence checking, we have introduced a PQE based algorithm that is complete and yet can fully exploit the similarity between the circuits to compare. Importantly, this algorithm cannot even be formulated without using the language of PQE.
- We showed that PQE can be used to generate "unexpected"properties of an implementation. These properties complement specification properties that this implementation is *supposed* to have. (Generation of a bad property means that the implementation is buggy.) We also showed that even the current version of *START* is good enough to generate properties for large designs.

## Directions For Future Research

- We will continue doing research in two directions: finding new applications of PQE and improving the quality of PQE solving.
- In particular, we will further study the application of PQE to property generation. As we mentioned earlier, PQE can be used to generate properties ranging from the weakest ones (equivalent to a single test) to the strongest property (specifying the "truth table"). So, property generation by PQE can complement and even, arguably, replace functional testing.
- To improve the quality of PQE-solving, we will keep working on *START* [8].
- One way to improve *START* is to reuse the learned clauses (called *certificates*). Currently, *START* only reuses the certificates learned in unsatisfiable subspaces (i.e. those where the formula is unsatisfiable). Reusing the certificates learned in satisfiable subspaces should dramatically boost the performance of *START*.
- One more way to make *START* much faster is to relax the constraint on the order in which variables are assigned. Currently, *START* may assign a quantified variable only if all *non-quantified* ones are already assigned.

## References


[1] A. Biere, A. Cimatti, E. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using sat procedures instead of bdds. In *DAC*, pages 317–320, 1999.

[2] A. R. Bradley. Sat-based model checking without unrolling. In *VMCAI*, pages 70–87, 2011.

[3] W. Craig. Three uses of the herbrand-gentzen theorem in relating model theory and proof theory. *The Journal of Symbolic Logic*, 22(3):269–285, 1957.

[4] E. Goldberg. Equivalence checking by logic relaxation. In *FMCAD-16*, pages 49–56, 2016.

[5] E. Goldberg. Property checking by logic relaxation. Technical Report arXiv:1601.02742 [cs.LO], 2016.

[6] E. Goldberg. Property checking without inductive invariant generation. Technical Report arXiv:1602.05829 [cs.LO], 2016.

[7] E. Goldberg. Generation of a complete set of properties. Technical Report arXiv:2004.05853 [cs.LO], 2020.

[8] E. Goldberg. Partial quantifier elimination by certificate clauses. Technical Report arXiv:2003.09667 [cs.LO], 2020.

[9] E. Goldberg and P. Manolios. Quantifier elimination by dependency sequents. In *FMCAD-12*, pages 34–44, 2012.

[10] E. Goldberg and P. Manolios. Quantifier elimination via clause redundancy. In *FMCAD-13*, pages 85–92, 2013.

[11] E. Goldberg and P. Manolios. Partial quantifier elimination. In *Proc. of HVC-14*, pages 148–164. Springer-Verlag, 2014.

[12] E. Goldberg and P. Manolios. Quantifier elimination by dependency sequents. *Formal Methods in System Design*, 45(2):111–143, 2014.

[13] A. Kuehlmann and F. Krohm. Equivalence Checking Using Cuts And Heaps. *DAC*, pages 263–268, 1997.

[14] M. Rabe. Incremental determinization for quantifier elimination and functional synthesis. In *CAV*, 2019.

[15] Berkeley Logic Synthesis and Verification Group. ABC: A system for sequential synthesis and verification, 2017. http://www.eecs.berkeley.edu/∼alanmi/abc.

[16] G. Tseitin. On the complexity of derivation in the propositional calculus. *Zapiski nauchnykh seminarov LOMI*, 8:234–259, 1968. English translation of this volume: Consultants Bureau, N.Y., 1970, pp. 115–125.

[17] CADET, https://github.com/MarkusRabe/cadet.

[18] A linux binary of start and some instances of pqe problems. http://eigold.tripod.com/software/start.tar.gz.