Structure-Aware Computing, Partial Quantifier Elimination And SAT

Eugene Goldberg

eu.goldberg@gmail.com

Abstract. Virtually all efficient algorithms of hardware verification are formula-specific i.e., take into account the structure of the formula at hand. So, those algorithms can be viewed as *structure-aware computing* (SAC). We relate SAC and *partial quantifier elimination* (PQE), a generalization of regular quantifier elimination. In PQE, one can take a *part* of the formula out of the scope of quantifiers. Interpolation can be viewed as a special case of PQE. The objective of this paper is twofold. First, we want to show that new powerful methods of SAC can be formulated in terms of PQE. We use three hardware verification problems (testing by property generation, equivalence checking and model checking) to explain how SAC is performed by PQE. Second, we want to demonstrate that PQE solving itself can benefit from SAC. To this end, we describe a new SAT procedure based on SAC and then use it to introduce a structure-aware PQE algorithm.

1 Introduction

Arguably, almost all efficient algorithms of hardware verification take into account the structure of the formula at hand i.e., they are formula-specific (see Appendix A). We will say that those algorithms employ *structure-aware computing*. In this paper, we relate structure-aware computing and partial quantifier elimination (PQE). Our objective here is twofold. First, we want to show off PQE as a language of structure-aware algorithms. Second, we want to demonstrate that PQE solving itself can benefit from being structure-aware.

In this paper, we consider only *propositional* formulas in *conjunctive normal form* (CNF) and only *existential* quantifiers. PQE is a generalization of regular quantifier elimination (QE) that is defined as follows [1]. Let $F(X, Y)$ be a quantifier-free formula where X, Y are sets of variables and G be a subset of clauses[1] of F. Given a formula $\exists X[F]$, the PQE problem is to find a quantifier-free formula $H(Y)$ such that $\exists X[F] \equiv H \wedge \exists X[F \setminus G]$. In contrast to *full* QE, only the clauses of G are taken out of the scope of quantifiers hence the name *partial* QE. Note that QE is just a special case of PQE where $G = F$ and the entire formula is unquantified. A key role in PQE solving plays *redundancy based*

---

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

reasoning: to take a set of clauses G out of $\exists X[F(X,Y)]$, one essentially needs to find a formula $H(Y)$ that makes G *redundant* in $H \wedge \exists X[F]$. The appeal of PQE is that it can be *much more efficient* than QE if G is a small piece of F. To solve PQE, one needs to make redundant only G whereas in QE the *entire* formula F is redundant in $H \wedge \exists X[F]$. (The complexity of PQE can even be reduced to linear [2].) So, it is beneficial to design algorithms based on PQE.

The idea of structure-aware computing by PQE is derived from the following observation. QE is a *semantic* operation in the sense that if $\exists X[F(X,Y)] \equiv H(Y)$ and $F' \equiv F$, then $\exists X[F'] \equiv H$. That is, to verify the correctness of H it suffices to know the truth table of F. On the other hand, PQE is a *structural* (i.e., formula-specific) operation in the following sense. The fact that $\exists X[F] \equiv H \wedge \exists X[F \setminus G]$ and $F' \equiv F$ **does not** imply that $\exists X[F'] \equiv H \wedge \exists X[F' \setminus G]$. For instance, G may be redundant in $\exists X[F]$ (and so $H \equiv 1$ and $\exists X[F] \equiv \exists X[F \setminus G]$) but not redundant in $\exists X[F']$. In other words, one cannot prove H correct using the truth table of F alone because H depends on the particulars of F. That is H is *formula-specific*. In this sense, PQE is similar to interpolation that is a structural operation *too* (see Section 4).

The contribution of this paper is threefold. *First*, we show that one can formulate powerful methods of structure-aware computing in terms of PQE. In particular, we show that the methods of property generation and equivalence checking by PQE published in [3,2] are actually examples of structure-aware computing (Section 5). Besides, we apply structure-aware computing by PQE to model checking (Section 6). *Second*, we demonstrate that PQE solving itself can benefit from taking into account the formula structure (Section 7). Namely, we introduce a structure-aware PQE algorithm called PQE^{sa} (Section 11). It is based on a structure-aware SAT algorithm we present before describing PQE^{sa} (Sections 8-10). The introduction of this algorithm is our *third* contribution.

The main body of this paper is structured as follows. (Some additional information is given in the appendix.) In Section 2, we provide basic definitions. A high-level view of PQE solving is presented in Section 3. As mentioned above, Sections 4-11 relate structure-aware computing, interpolation, PQE, and SAT. In Section 12, we make conclusions.

2 Basic Definitions

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

Definition 1. *We assume that formulas have only Boolean variables. A **literal** of a variable v is either v (the positive literal) or its negation $\overline{v}$ (the negative literal). 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** unless otherwise stated.*

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

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

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

Definition 6. *Given a formula $\exists X[F]$, a clause C of F is called $\textbf{quantified}$ if $Vars(C) \cap X \neq \emptyset$. Otherwise, the clause C is called $\textbf{unquantified}$.*

Definition 7. *Let G, H be formulas that may have existential quantifiers. We say that G, H are $\textbf{equivalent}$, written $\textbf{G} \equiv \textbf{H}$, if $G_{\vec{q}} = H_{\vec{q}}$ for all full assignments $\vec{q}$ to $Vars(G) \cup Vars(H)$.*

Definition 8. *Let $F(X, Y)$ be a formula and $G \subseteq F$ and $G \neq \emptyset$. The clauses of G are said to be $\textbf{redundant in}$ $\exists\textbf{X}[\textbf{F}]$ if $\exists X[F] \equiv \exists X[F \setminus G]$. If $F \setminus G$ implies G, the clauses of G are redundant in $\exists X[F]$ but the reverse is not true.*

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

Example 1. Consider formula $F = C_1 \wedge \cdots \wedge C_5$ where $C_1 = \overline{x}_3 \vee x_4$, $C_1 = y_1 \vee x_3$, $C_2 = y_1 \vee \overline{x}_4$, $C_4 = y_2 \vee x_4$, $C_5 = y_2 \vee \overline{x}_4$. Let $Y = \{y_1, y_2\}$ and $X = \{x_3, x_4\}$. Consider the PQE problem of taking C_1 out of $\exists X[F]$ i.e., finding $H(Y)$ such that $\exists X[F] \equiv H \wedge \exists X[F \setminus \{C_1\}]$. In Subsection 3.3 we show that $\exists X[F] \equiv y_1 \wedge \exists X[F \setminus \{C_1\}]$ i.e., $H = y_1$ is a solution to this PQE problem.

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 $\textbf{resolvable}$ on w. Let C be the clause consisting of the literals of C' and C'' minus those of w. Then C is said to be obtained by $\textbf{resolution}$ of C' and C'' on w.*

Definition 11. *Let C be a clause of a formula F and $w \in Vars(C)$. The clause C is called $\textbf{blocked}$ in F at w [4] if no clause of F is resolvable with C on w.*

Proposition 1. *Let a clause C be blocked in a formula $F(X,Y)$ at a variable $x \in X$. Then C is redundant in $\exists X[F]$, i.e., $\exists X[F \setminus \{C\}] \equiv \exists X[F]$.*

Proposition 1 was proved in [2]. The proofs of all propositions (old and new ones) are given in Appendix B.

3 PQE solving

In this section, we briefly describe the PQE algorithm called *DS-PQE* [1]. Our objective here is to provide an idea of how the PQE problem can be solved. So, in Subsection 3.2, we give a high-level description of this algorithm and in Subsection 3.3 we present an example of PQE solving. In Sections 7 and 11 we continue the topic of PQE solving.

3.1 Some background

Information on QE in propositional logic can be found in [5,6,7,8,9,10]. QE by redundancy based reasoning is presented in [11,12]. One of the merits of such reasoning is that it allows to introduce *partial* QE. A description of PQE algorithms and their sources can be found in [1,2,13,14,15].

3.2 High-level view

Like all existing PQE algorithms, *DS-PQE* uses *redundancy based reasoning* justified by the proposition below.

Proposition 2. *Formula $H(Y)$ is a solution to the PQE problem of taking G out of $\exists X[F(X,Y)]$ (i.e., $\exists X[F] \equiv H \land \exists X[F \setminus G]$) iff*

1. $F \Rightarrow H$ *and*
2. $H \land \exists X[F] \equiv H \land \exists X[F \setminus G]$

So, to take G out of $\exists X[F(X,Y)]$, it suffices to find $H(Y)$ implied by F that makes G *redundant* in $H \land \exists X[F]$. We refer to clauses of G as **target** ones.

Below, we provide some basic facts about *DS-PQE*. Since taking out an un-quantified clause is trivial, we assume that the formula G contains only *quantified* clauses. *DS-PQE* finds a solution to the PQE problem above by branching on variables of F. The idea here is to reach a subspace $\vec{q}$ where every clause of G can be easily proved or made redundant in $\exists X[F]$. *DS-PQE* branches on un-quantified variables, i.e., those of Y, *before* quantified ones. Like a SAT-solver, *DS-PQE* runs Boolean Constraint Propagation (BCP). If a conflict occurs in subspace $\vec{q}$, *DS-PQE* generates a conflict clause C_{cnfl} and adds it to F to *make* clauses of G redundant in subspace $\vec{q}$. *DS-PQE* can also prove that G is *already* redundant in a subspace *without* reaching a conflict and adding a conflict clause.

To express the redundancy of a clause C in a subspace $\vec{q}$, *DS-PQE* uses a record $\vec{q} \to C$ called a **D-sequent** [11,12] (see below). It states the redundancy

of C in the current formula $\exists X[F]$ in subspace $\vec{q}$. This D-sequent also holds in any formula $\exists X[F^*]$ where F^* is obtained from F by adding clauses implied by F. A D-sequent derived for a target clause C is called **atomic** if the redundancy of C can be trivially proved. D-sequents derived in different branches can be resolved similarly to clauses. For every target clause C of the original formula G, *DS-PQE* uses such resolution to eventually derive the D-sequent $\emptyset \to C$. The latter states the redundancy of C in the *entire* space. At this point *DS-PQE* terminates. The solution $H(Y)$ to the PQE problem found by *DS-PQE* consists of the *unquantified* clauses added to the initial formula F to make G redundant.

3.3   An example of PQE solving

Here we show how *DS-PQE* solves Example 1 introduced in Section 2. Recall that one takes $G = \{C_1\}$ out of $\exists X[F(X,Y)]$ where $F = C_1 \wedge \cdots \wedge C_5$ and $C_1 = \bar{x}_3 \vee x_4$, $C_2 = y_1 \vee x_3$, $C_3 = y_1 \vee \bar{x}_4$, $C_4 = y_2 \vee x_4$, $C_5 = y_2 \vee \bar{x}_4$ and $Y = \{y_1, y_2\}$ and $X = \{x_3, x_4\}$. One needs to find $H(Y)$ such that $\exists X[F] \equiv H \wedge \exists X[F \setminus \{C_1\}]$.

   Assume *DS-PQE* picks the variable y_1 for branching and first explores the branch $\vec{q}' = (y_1 = 0)$. In subspace $\vec{q}'$, clauses C_2, C_3 become unit. (An unsatisfied clause is called *unit* if it has only one unassigned literal.) After assigning $x_3 = 1$ to satisfy C_2, the clause C_1 turns into unit too and a conflict occurs (to satisfy C_1 and C_3, one has to assign the opposite values to x_4). After a standard conflict analysis [16], a conflict clause $C_{cnfl} = y_1$ is obtained by resolving C_2 and C_3 with C_1. To *make* C_1 redundant in subspace $\vec{q}'$, *DS-PQE* adds C_{cnfl} to F. The redundancy of C_1 is expressed by the D-sequent $\vec{q}' \to C_1$. This D-sequent is an example of an *atomic* one. It asserts that C_1 is redundant in subspace $\vec{q}'$.

   Having finished the first branch, *DS-PQE* considers the second branch: $\vec{q}'' = (y_1 = 1)$. Since the clause C_2 is satisfied by $\vec{q}''$, no clause of F is resolvable with C_1 on variable x_3 in subspace $\vec{q}''$. Hence, C_1 is blocked at variable x_3 and thus redundant in $\exists X[F]$ in subspace $\vec{q}''$. So, *DS-PQE* generates the D-sequent $\vec{q}'' \to C_1$. This D-sequent is another example of an *atomic* D-sequent. It states that C_1 is *already* redundant in $\exists X[F]$ in subspace $\vec{q}''$ (without adding a new clause). Then *DS-PQE* resolves the D-sequents $(y_1 = 0) \to C_1$ and $(y_1 = 1) \to C_1$ on y_1. This resolution produces the D-sequent $\emptyset \to C_1$ stating the redundancy of C_1 in $\exists X[F]$ in the *entire* space (i.e., globally). Recall that $F_{fin} = C_{cnfl} \wedge F_{init}$ where F_{fin} and F_{init} denote the final and initial formula F respectively. That is C_{cnfl} is the only unquantified clause added to F_{init}. So, *DS-PQE* returns C_{cnfl} as a solution $H(Y)$. The clause $C_{cnfl} = y_1$ is indeed a solution since it is implied by F_{init} and C_1 is redundant in $C_{cnfl} \wedge \exists X[F_{init}]$. So both conditions of Proposition 2 are met and thus $\exists X[F_{init}] \equiv y_1 \wedge \exists X[F_{init} \setminus \{C_1\}]$.

4   PQE And Interpolation

Interpolation [17,18] is an example of structure-aware computing. In this section, we show that interpolation can be viewed as a special case of PQE.

Let $A(X, Y) \wedge B(Y, Z)$ be an unsatisfiable formula where X, Y, Z are sets of variables. Let $I(Y)$ be a formula such that $A \wedge B \equiv I \wedge B$ and $A \Rightarrow I$. Replacing $A \wedge B$ with $I \wedge B$ is called *interpolation* and I is called an *interpolant*. PQE is similar to interpolation in the sense that the latter is a *structural* rather than semantic operation. Suppose, for instance, that $A'(X, Y) \wedge B$ is a formula such that $A' \wedge B \equiv A \wedge B$ but $A' \not\equiv A$. Then, in general, the formula I above *is not* an interpolant for $A' \wedge B$ i.e., $A' \wedge B \not\equiv I \wedge B$ or $A' \not\Rightarrow I$.

Now, let us describe interpolation in terms of PQE. Consider the formula $\exists W[A \wedge B]$ where $W = X \cup Z$ and A, B are the formulas above. Let $A^*(Y)$ be obtained by taking A out of the scope of quantifiers i.e., $\exists W[A \wedge B] \equiv A^* \wedge \exists W[B]$. Since $A \wedge B$ is unsatisfiable, $A^* \wedge B$ is unsatisfiable too. So, $A \wedge B \equiv A^* \wedge B$. If $A \Rightarrow A^*$, then A^* is an interpolant.

The *general case* of PQE that takes A out of $\exists W[A \wedge B]$ is different from the instance above in three aspects. First, $A \wedge B$ can be *satisfiable*. Second, one does not assume that $Vars(B) \subset Vars(A \wedge B)$. That is, in general, PQE *is not meant* to produce a new formula with a smaller set of variables. Third, a solution A^* is generally implied by $A \wedge B$ rather than by A *alone*. So, interpolation can be viewed as a special case of PQE. Hence, one can expect PQE to enable a much more general set of structure-aware algorithms than interpolation.

5 Previous Results As Structure-Aware Computing

In this section, we recall the methods of property generation and equivalence checking by PQE [2,3]. We describe them by the example of combinational circuits. Our objective here is to show that these are actually powerful methods of *structure-aware computing*.

5.1 Representing a combinational circuit by a CNF formula

Let $M(X, V, W)$ be a combinational circuit where X, V, W are sets of 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 Tseitsin's transformations [19]. Namely, $F = F_{g_1} \wedge \cdots \wedge F_{g_k}$ 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 variable and x_1, x_2 denote the input variables. 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.

5.2 Testing, property generation, and structure-aware computing

Testing is a workhorse of functional verification. The appeal of testing is that it is surprisingly effective in bug hunting taking into account that the set of tests makes up only a tiny part of the truth table. This effectiveness can be attributed to the fact that modern procedures are aimed at testing a particular *implementation* rather than sampling the truth table. So, testing can be viewed as an instance of structure-aware computing. In this subsection, we recall property generation by PQE [2] that is a *generalization* of testing. We show that using PQE dramatically boosts the power of structure-aware computing.

In addition to incompleteness, testing has the following flaw. Let $M(X, V, W)$ be a combinational circuit and F be a formula specifying M as described above. Let $\vec{v}$ denote a single test i.e., a full assignment to V. The input/output behavior corresponding to $\vec{v}$ can be cast as a property $H^{\vec{v}}(V, W)$ of M (i.e., F implies $H^{\vec{v}}$, see Appendix C). If the test $\vec{v}$ exposes a bug, then $H^{\vec{v}}$ is an *unwanted* property of M. The flaw above is that $H^{\vec{v}}$ is a **weakest** property of M. So, testing can overlook a bug easily exposed by a *stronger* unwanted property (e.g., a property stating that some valid combination of output values is never produced by M). A comprehensive solution would be to generate the truth table $T(V, W)$, which is the *strongest* property of M. (T can be produced by performing QE on $\exists X[F]$ i.e., $T \equiv \exists X[F]$.) However, computing T can be prohibitively expensive. PQE allows to produce properties that are much stronger than single-test properties $H^{\vec{v}}$ but can be generated much more efficiently than the truth table T.

For the sake of simplicity, consider property generation by taking a single clause C out of $\exists X[F]$. Let $H(V, W)$ be a solution, i.e., $\exists X[F] \equiv H \wedge \exists X[F \setminus \{C\}]$. Since $F \Rightarrow H$, the solution H is a *property* of M. If H is an *unwanted* property, M has a bug. If taking out C is still too hard, one can simplify the problem by clause *splitting*. The idea here is to replace C with clauses $C \vee p$ and $C \vee \overline{p}$ where $p \in \mathit{Vars}(F)$ and take out, say, $C \vee p$ instead of C. Then PQE becomes simpler but produces a weaker property H. Given a single test $\vec{v}$, one can produce the single-test property $H^{\vec{v}}$ by combining PQE with splitting C on *all* input variables (see Appendix C).

Like testing, property generation is an instance of **structure-aware computing**. Indeed, let M' be a circuit equivalent to M but having a different structure and F' be a formula specifying M'. Let H be a property obtained by taking a *single* clause out of $\exists X[F]$. Intuitively, to produce H (or a close stronger property) from $\exists X'[F']$, one may need to take out a *large* set of clauses. That is H is much easier to obtain and hence more natural for M than for M'.

Here some experimental results on property generation reported in [2] that describe the status quo. Those results were produced by an optimized version of *DS-PQE* [2]. The latter was used to generate properties for the combinational circuit M_k obtained by unfolding a sequential circuit N for k time frames. Those properties were employed to produce invariants of N. A sample of HWMCC benchmarks containing from 100 to 8,000 latches was used in those experiments. *DS-PQE* managed to generate a lot of properties of M_k that turned out to be

invariants of N. *DS-PQE* also successfully generated an *unwanted* invariant of a tailor-made FIFO buffer and so identified a hard-to-find bug.

5.3 Equivalence checking and structure-aware computing

In this subsection, we discuss equivalence checking by PQE [3] in the context of structure-aware computing. Let $M'(X', V', w')$ and $M''(X'', V'', w'')$ be the single-output combinational circuits to check for equivalence. Here X^α, V^α are the sets of internal and input variables and w^α is the output variable of M^α where $\alpha \in \{', ''\}$. Circuits M', M'' are called *equivalent* if they produce identical values of w', w'' for identical inputs $\vec{v}', \vec{v}''$ (i.e., identical full assignments to V', V'').

Let $F'(X', V', w')$ and $F''(X'', V'', w'')$ specify M' and M'' respectively as described in Subsection 5.1. Let F^* denote the formula $F'(X', V, w') \wedge F''(X'', V, w'')$ where $V' = V'' = V$. (V' and V'' are identified in F^* because equivalence checking is concerned only with identical assignments to V' and V''.) Let $Z^* = X' \cup X'' \cup V$. A straightforward but hugely inefficient method of equivalence checking is to perform QE on $\exists Z^*[F^*]$ to derive $w' \equiv w''$. The most efficient equivalence checking tools use the method that we will call **cut propagation (*CP*)** [20,21,22]. The *CP* method can be viewed as an efficient *approximation* of QE meant for proving equivalence of circuits that are *very similar*.

The idea of *CP* is to build a sequence of cuts $Cut_1, \ldots, Cut_k$ of M' and M'' and find cut points of M', M'' for which some simple *pre-defined* relations Rel_1, $\ldots$, Rel_k hold (e.g., functional equivalence). Computations move from inputs to outputs where $Cut_1 = V$ and $Cut_k = \{w', w''\}$. The relation Rel_1 is *set* to the constant 1 whereas $Rel_2, \ldots, Rel_k$ are *computed* in an inductive manner. (That is Rel_i is obtained using previously derived $Rel_1, \ldots, Rel_{i-1}$.) The objective of *CP* is to prove $Rel_k = (w' \equiv w'')$. The main flaw of *CP* is that circuits M' and M'' may not have cut points related by pre-defined relations even if M' and M'' are very similar. In this case *CP* fails. So, it is *incomplete* even for similar circuits M', M''. Despite its flaws, *CP* is a successful practical structure-aware method that exploits the similarity of M' and M''.

In [3], a method of equivalence checking based on PQE was introduced. This method also uses cut propagation. So, we will refer to it as CP^{pqe}. Let F denote the formula $F'(X', V', w') \wedge F''(X'', V'', w'')$ where V' and V'' are *separate* sets. Let $Eq(V', V'')$ denote a formula such that $Eq(\vec{v}', \vec{v}'') = 1$ iff $\vec{v}' = \vec{v}''$. Let $Z = X' \cup X'' \cup V' \cup V''$. CP^{pqe} is based on the proposition below proved in [3].

Proposition 3. *Assume M', M'' do not implement a constant (0 or 1). Let $\exists Z[Eq \wedge F] \equiv H(w', w'') \wedge \exists Z[F]$. Then M' and M'' are equivalent iff $H \Rightarrow (w' \equiv w'')$.*

Hence, to find if M', M'' are equivalent, it suffices to take Eq out of $\exists Z[Eq \wedge F]$. (Checking if M' or M'' implement a constant reduces to a few simple SAT checks.) Note that $Eq(V', V'') \wedge F(\ldots, V', V'', \ldots)$ is semantically the same as the formula $F^*(\ldots, V, \ldots)$ above used in *CP*. However, CP^{pqe} *cannot* be formulated in terms of F^* since it exploits the *structure* of $Eq \wedge F$ (i.e., the presence of Eq).

CP^{pqe} takes Eq out of $\exists Z[Eq \wedge F]$ incrementally, cut by cut, by computing relations Rel_i (see Fig. 1). Rel_1 is set to Eq whereas the remaining relations Rel_i are computed by PQE (see Appendix D). CP^{pqe} provides a *dramatically more powerful* version of **structure-aware computing** than CP. (One can show that CP is just a special case of CP^{pqe} where only particular relationships between cut points are considered.) The difference between CP^{pqe} and CP is threefold. First, there are *no pre-defined relationships* to look for. Relations Rel_i just need to satisfy a very simple property: $\exists Z[Rel_{i-1} \wedge Rel_i \wedge F] \equiv \exists Z[Rel_i \wedge F]$. That is the next relation Rel_i makes the previous relation Rel_{i-1} *redundant*. Second, CP^{pqe} is *complete*. Third, as formally proved in [3], relations Rel_i become quite *simple* if M' and M'' are structurally similar.
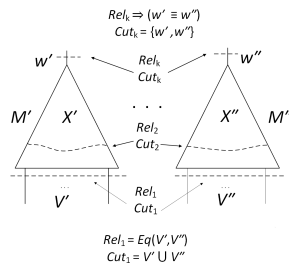

Fig. 1: Proving equivalence by the CP^{pqe} method

In [3], some experiments with CP^{pqe} were described where circuits M', M'' containing a multiplier of various sizes were checked for equivalence. (The size of the multiplier ranged from 10 to 16 bits. An optimized version of $DS\text{-}PQE$ was used as a PQE solver.) M', M'' were intentionally designed so that they were structurally similar but did not have any functionally equivalent points. A high-quality tool called ABC [23] showed very poor performance, whereas CP^{pqe} solved all examples efficiently. In particular, CP^{pqe} solved the example involving a 16-bit multiplier in 70 seconds, whereas ABC failed to finish it in 6 hours.

6 Model Checking And Structure-Aware Computing

In this section, we apply structure-aware computing by PQE to finding the reachability diameter, i.e., to a problem of model checking.

6.1 Motivation and some background

An efficient algorithm for finding the reachability diameter can be quite beneficial. Suppose one knows that the reachability diameter of a sequential circuit N is less than k. Then, to verify *any* invariant of N, it suffices to check if it holds for the states of N reachable in at most $k-1$ transitions. This check can be done by bounded model checking [24]. Finding the reachability diameter of a sequential circuit by existing methods essentially requires computing the set of all reachable states [25,26], which does not scale well. An upper bound on the reachability diameter called the recurrence diameter can be found by a SAT-solver [27]. However, this upper bound is very imprecise. Besides, its computing does not scale well either.

6.2 Some definitions

Let S specify the set of state variables of a sequential circuit N. Let $T(S', S'')$ denote the transition relation of N where S', S'' are the sets of present and next state variables. Let formula $I(S)$ specify the initial states of N. (A **state** is a full assignment to S.) A state $\vec{s}_{k+1}$ of N with initial states I is called **reachable** in k transitions if there is a sequence of states $\vec{s}_1, \ldots, \vec{s}_{k+1}$ such that $I(\vec{s}_1) = 1$ and $T(\vec{s}_i, \vec{s}_{i+1}) = 1$, $i = 1, \ldots, k$. For the reason explained in Remark 1, we assume that N can **stutter**. That is, $T(\vec{s}, \vec{s}) = 1$ for every state $\vec{s}$. (If N lacks stuttering, it can be easily introduced.)

Remark 1. If N can stutter, the set of states of N reachable in k transitions is the same as the set of states reachable in *at most* k transitions. This nice property holds because, due to the ability of N to stutter, each state reachable in p transitions is also reachable in k transitions where $k > p$.

Let $R_k(S)$ be a formula specifying the set of states of N reachable in k transitions. That is $R_k(\vec{s}) = 1$ iff $\vec{s}$ is reachable in k transitions. Let S_i specify the state variables of i-th time frame. Formula $R_k(S)$ where $S = S_{k+1}$ can be computed by performing QE on $\exists S_{1,k}[I_1 \wedge T_{1,k}]$. Here $\boldsymbol{S_{1,k}} = S_1 \cup \cdots \cup S_k$ and $\boldsymbol{T_{1,k}} = T(S_1, S_2) \wedge \cdots \wedge T(S_k, S_{k+1})$. We will call $Diam(N, I)$ the **reachability diameter** of N with initial states I if any reachable state of N requires at most $Diam(N, I)$ transitions to reach it.

6.3 Computing reachability diameter

In this subsection, we consider the problem of deciding if $Diam(N, I) < k$. A straightforward way to solve this problem is to compute R_{k-1} and R_k by performing QE as described above. $Diam(N, I) < k$ iff R_{k-1} and R_k are equivalent. Unfortunately, computing R_{k-1} and R_k even for a relatively small value of k can be very hard or simply infeasible for large circuits. Below, we show that one can arguably solve this problem more efficiently using *structure-aware computing*.

Proposition 4. *Let $k \geq 1$. Let $\exists S_{1,k}[I_1 \wedge I_2 \wedge T_{1,k}]$ be a formula where I_1 and I_2 specify the initial states of N in terms of variables of S_1 and S_2 respectively. Then $Diam(N, I) < k$ iff I_2 is redundant in $\exists S_{1,k}[I_1 \wedge I_2 \wedge T_{1,k}]$.*

Proposition 4 reduces checking if $Diam(N, I) < k$ to the *decision version* of PQE (i.e., finding if I_2 is redundant in $\exists S_{1,k}[I_1 \wedge I_2 \wedge T_{1,k}]$). Note that the presence of I_2 simply "cuts out" the initial time frame (indexed by 1). So, *semantically*, $\exists S_{1,k}[I_1 \wedge I_2 \wedge T_{1,k}]$ is the same as $\exists S_{2,k}[I_2 \wedge T_{2,k}]$ i.e., specifies the states reachable in $k-1$ transitions. But the former has a different *structure* one can exploit by PQE. Namely, proving I_2 redundant in $\exists S_{1,k}[I_1 \wedge I_2 \wedge T_{1,k}]$ means that the sets of states reachable in $k-1$ and k transitions (the latter specified by $\exists S_{1,k}[I_1 \wedge T_{1,k}]$) are identical. Importantly, I_2 is a *small* piece of the formula. So, proving it redundant can be much more efficient than computing R_{k-1} and R_k. For instance, computing R_k by QE requires proving the *entire formula* $I_1 \wedge T_{1,k}$ redundant in $R_k \wedge \exists S_{1,k}[I_1 \wedge T_{1,k}]$.

7 Making PQE Solving More Structure-Aware

So far, we have considered using PQE for creating new methods of structure-aware computing. However, PQE solving *itself* can benefit from being structure-aware. In this section, we describe a problem of the current methods of PQE solving by the example of *DS-PQE* presented in Section 3. We argue that this problem is caused by changing the formula via adding *quantified* conflict clauses. We address this problem in two steps. First, in Sections 8-10 we describe a structure-aware SAT algorithm called SAT^{sa} that *separates* the original clauses from proof ones (here 'sa' stands for 'structure-aware'). So, SAT^{sa} enjoys the power of learning proof clauses while preserving the original formula *intact*. Second, in Section 11 we use the idea of SAT^{sa} to introduce a structure-aware PQE algorithm where the problem above is fixed.

Below we explain the *problem* with adding new quantified clauses (i.e., those with quantified variables) in more detail. Consider, the PQE problem of taking a set of clauses G out of $\exists X[F(X,Y)]$. *DS-PQE* does this by branching on variables of F until the target clauses are proved/made redundant in the current subspace $\vec{q}$. Then the results of different branches are merged. If a conflict occurs in subspace $\vec{q}$, the target clauses are *made* redundant by adding a conflict clause C_{cnfl} that is falsified by $\vec{q}$. Otherwise, *DS-PQE* simply shows that the target clauses are already redundant in subspace $\vec{q}$ without adding any clauses. The addition of conflict clauses is inherited by *DS-PQE* from modern CDCL solvers where CDCL stands for "Conflict Driven Clause Learning".

On one hand, adding conflict clauses to F increases the power of BCP. So, it helps to prove redundancy of a target clause C in a subspace $\vec{q}$ if $(F \setminus \{C\}) \Rightarrow C$ in this subspace [1]. Namely, if C is satisfied by an assignment derived from a clause B that is unit in subspace $\vec{q}$ then $B \Rightarrow C$ in this subspace. Note that in this case C is redundant *regardless* of whether formula F is quantified.

On the other hand, adding conflict clauses makes it harder to prove redundancy displaying itself only in quantified formulas. Suppose that a target clause C is redundant in $\exists X[F]$ in subspace $\vec{q}$ *not being implied* in this subspace by $F \setminus \{C\}$. *DS-PQE* proves this type of redundancy as follows [1]. Let x be a quantified variable of C. *DS-PQE* tries to show that all clauses of F resolvable with C on x are redundant in subspace $\vec{q}$. If so, C is blocked at x in subspace $\vec{q}$ and so redundant in $\exists X[F]$ in this subspace. Note that a quantified *conflict clause* can be resolvable with C on x. So, adding conflict clauses **increases** the number of clauses one needs to prove redundant to show that C is blocked at x. This makes proving redundancy of C **harder**.

8 SAT Solving By Structure-Aware Computing

In Sections 9 and 10, we describe a structure-aware SAT solver called SAT^{sa}. In Section 11, we use the idea of SAT^{sa} to introduce a structure-aware PQE algorithm. In this section, we give some background and list attractions of SAT^{sa}.

8.1 Some background

Given a formula $F(X)$, SAT is to check if F is satisfiable i.e., whether $\exists X[F]{=}1$. SAT plays a huge role in practical applications. Modern CDCL solvers are descendants of the DPLL procedure [28] that checks the satisfiability of F by looking for an explicit satisfying assignment. They identify subspaces where F is unsatisfiable due to an assignment conflict and learn conflict clauses to avoid those subspaces (see e.g., [16,29,30,31]). The conflict clauses are derived by resolution. Importantly, the basic operation of the DPLL procedure is variable splitting (i.e., exploring the branches $x = 0$ and $x = 1$ for a variable $x \in X$). This operation is *semantic*. Intuitively, a *formula-specific* SAT solver can be more efficient than semantic since the latter is, in a way, an "overkill".

8.2 Some attractions of structure-aware SAT solving

SAT^{sa} has at least two attractions. **The first attraction** is that SAT^{sa} separates the initial and learned clauses thus keeping the original formula intact. Besides, SAT^{sa} is "clause-oriented", which simplifies moving between different parts of the formula. So, SAT^{sa} can be tuned to a particular class of formulas. Suppose, for instance, that a formula F specifies equivalence checking of similar combinational circuits M' and M''. When checking the satisfiability of F it is useful to compute relations between cut points of M', M'' for a sequence of cuts (see Subsection 5.3). Such computations can be mimicked by SAT^{sa} (see Appendix G) but are hard for a CDCL solver for two reasons. First, adding conflict clauses to F *blurs* cuts. Second, CDCL solvers do not have a natural way of moving from one part of the formula to another.

Separation of the original and learned clauses is easy in SAT^{sa} because the proof generated by SAT^{sa} is a collection of so-called *proof clauses*. A proof clause certifies that the current subspace $\vec{q}$ does not have an assignment satisfying F that also satisfies only one literal of a particular clause of F. (A clause falsified by $\vec{q}$ that is derived by a CDCL solver can be viewed as an "overkill" proof clause refuting the existence of *any* satisfying assignment in subspace $\vec{q}$.)

The second attraction of SAT^{sa} is that the proof clauses mentioned above are derived not only by resolution but also by structural derivations enabled by Proposition 6 below. In general, a structural derivation cannot be trivially simulated by resolution. However, as we show in Example 4, simulation of structural derivation can be simplified if resolution is enhanced with adding blocked clauses [32]. This is important for two reasons. First, the ability to add blocked clauses makes resolution exponentially more powerful. So, structural derivations give SAT^{sa} extra power in comparison to a solver based on resolution *alone*. Second, enhancing a regular CDCL solver by the ability to add blocked clauses is a tall order because there are too many blocked clauses to choose from. (So, it is virtually impossible to *efficiently* find blocked clauses that are useful.) SAT^{sa} *avoids* this problem since it taps to the power of blocked clauses without generating them *explicitly*.

9   Propositions Supporting SAT^{sa}

In this section, we present two propositions on which SAT^{sa} is based. These propositions strengthen the results of [33].

Proposition 5. *Let C be a clause of a formula $F(X)$. If F is satisfiable, there exists an assignment $\vec{x}$ satisfying F that a) satisfies only one literal of C or b) satisfies only one literal l of another clause $C' \in F$ where l is present in C.*

Proposition 6. *Let S be a formula falsified by every assignment meeting the condition a) or b) of Proposition 5 and $F \Rightarrow S$. Then F is unsatisfiable.*

The satisfiability of the formula S in Proposition 6 is irrelevant. In particular, it can be satisfiable (see Example 4). This means that the proof of unsatisfiability by Proposition 6 is **formula-specific**: in general, if a formula F implies a satisfiable formula, this does not entail that F is unsatisfiable. If Proposition 6 holds in a subspace $\vec{r}$, a clause B implied by F and falsified by $\vec{r}$ can be generated. Similarly to a conflict clause, B consists of literals falsified by the *relevant* value assignments of $\vec{r}$. (Subsection 10.2 and Appendix E give more details.)

One can construct S as a conjunction of l-proof clauses defined below.

Definition 12. *Let C be a clause of $F(X)$ and l be a literal of C. The set of full assignments to X satisfying only the literal l is called the **l-vicinity** of C. We will say that the shortest assignment $\vec{q}$ satisfying l and falsifying the other literals of C **specifies** the l-vicinity of C. A clause B is called an **l-proof** for C if $\vec{q}$ falsifies B. If $F \Rightarrow B$, the l-vicinity of C has no assignment satisfying F.*

Example 3. Let $C = x_1 \vee x_2 \vee x_3$ be a clause of F. Then, say, the x_1-vicinity of C is specified by the assignment $\vec{q} = (\boldsymbol{x_1 = 1}, x_2 = 0, x_3 = 0)$. (The single-variable assignment satisfying C is shown in bold.) The clause $B = \overline{x}_1 \vee x_2 \vee x_3$ is an example of an x_1-proof clause for C. If $F \Rightarrow B$, the subspace $\vec{q}$ has no assignment satisfying F. Clauses $x_2 \vee x_3$ and $\overline{x}_1$ are also x_1-proof clauses for C.

Definition 13. *We will refer to the clause C of Proposition 5 as the **primary** clause and the clauses of F sharing literals with C as the **secondary** clauses.*

Example 4. Let F be equal to $C_1 \wedge C_2 \wedge C_3 \wedge \dots$ where $C_1 = x_1 \vee \overline{x}_2$, $C_2 = x_1 \vee x_5$ and $C_3 = \overline{x}_2 \vee \overline{x}_6 \vee x_8$. Assume that C_1, C_2, C_3 are the only clauses of F with literals $x_1, \overline{x}_2$ whereas F can contain any number of clauses with literals $\overline{x}_1, x_2$. Let us apply Proposition 6 using C_1 as the primary clause. Then C_2, C_3 are the secondary clauses. So, to prove F unsatisfiable it suffices to show that the x_1-vicinity and $\overline{x}_2$-vicinity of C_1, and the x_1-vicinity of C_2 and $\overline{x}_2$-vicinity of C_3 do not contain a satisfying assignment. That is there is no such an assignment in subspaces $(\boldsymbol{x_1 = 1}, x_2 = 1)$, $(x_1 = 0, \boldsymbol{x_2 = 0})$ and $(\boldsymbol{x_1 = 1}, x_5 = 0)$ and $(\boldsymbol{x_2 = 0}, x_6 = 1, x_8 = 0)$.

Assume, for the sake of clarity, that S is the formula consisting of the proof clauses obtained by flipping the literal x_1 or $\overline{x}_2$ of C_1, the literal x_1 of C_2 and $\overline{x}_2$ of C_3. That is $S = C_1' \vee C_1'' \vee C_2' \vee C_3'$ where $C_1' = \overline{x}_1 \vee \overline{x}_2$, $C_1'' = x_1 \vee x_2$,

$C_2' = \overline{x}_1 \vee x_5$ and $C_3' = x_2 \vee \overline{x}_6 \vee x_8$. To prove F unsatisfiable it suffices to show that $F \Rightarrow S$. Note that S is a satisfiable formula. It remains satisfiable even if it is conjoined with the primary and secondary clauses i.e., C_1, C_2, C_3. Denote the formula $S \wedge C_1 \wedge C_2 \wedge C_3$ as S^*. The assignments satisfying S^* make up subspaces $(x_1 = 1, x_2 = 0, x_5 = 1, x_6 = 0)$ and $(x_1 = 1, x_2 = 0, x_5 = 1, x_8 = 1)$ and satisfy at least two literals of C_1, C_2, C_3. According to Proposition 5, such assignments can be ignored when proving F unsatisfiable. Note that the clause $K = \overline{x}_1 \vee x_2 \vee \overline{x}_5$ is *blocked* in F at x_1 (since K is unresolvable with C_1 and C_2 on x_1). Adding K to S^* makes the latter unsatisfiable, which can be easily proved by resolution. However, the *derivation of K* from F by resolution alone can be hard.

Remark 2. Similarly to variable splitting of CDCL solvers, Proposition 6 partitions the original problem into simpler subproblems (of examining l-vicinities of some clauses). The difference here is that variable splitting is *semantic* whereas problem partitioning by Proposition 6 is *formula-specific* i.e., structural.

10   Description Of SAT^{sa}

As we mentioned earlier, one of the attractions of SAT^{sa} is the ability to tune to a specific class of formulas (an example with equivalence checking is given in Appendix G). In this section, we give a *generic* version of SAT^{sa} meant to be a starting point for *formula-specific* implementations. It can also be used in the quest for an efficient *general-purpose* SAT algorithm that can exploit the structure of the formula. So, in this section, we just identify a *direction* to pursue. We believe that due to the novelty and promise of this direction, even the description of a generic version of SAT^{sa} is of high value.

10.1   Top procedure of SAT^{sa}

SAT^{sa} accepts the formula F to check for satisfiability, the current set P of proof clauses, an assignment $\vec{q}$ specifying the current subspace (see Fig. 2). We assume that P is an in/out parameter. That is, if P is changed by a procedure, the caller of this procedure gets the changed value of P. SAT^{sa} recursively calls itself, the initial call being $SAT^{sa}(F, \emptyset, \emptyset)$. Importantly, F *does not change*, whereas P grows. SAT^{sa} returns a satisfying assignment (if F is satisfiable in subspace $\vec{q}$) or a clause falsified by $\vec{q}$ (if F is unsatisfiable in subspace $\vec{q}$). A recursive invocation of SAT^{sa} can also return a clause U that is unit in subspace $\vec{q}$. Then the satisfiability of F in subspace $\vec{q}$ remains undecided.

$\quad$ SAT^{sa} runs a *while* loop that is broken if some conditions are met. SAT^{sa} starts an iteration of this loop by running BCP on $F \cup P$ that assigns the new variables of $Vars(\vec{q})$ i.e., those that are assigned in $\vec{q}$ but not in $F \cup P$ yet (line 2). Besides, as usual, BCP makes implied assignments satisfying unit clauses. If BCP leads to a conflict, a conflict clause C_{cnfl} is generated [16] and returned by SAT^{sa} (line 3). If, by extending $\vec{q}$ with implied assignments, BCP produces an assignment satisfying F, then SAT^{sa} returns $\vec{q}$ (line 4). If SAT^{sa} does not

terminate after BCP, it picks a clause $C \in F$ not satisfied by $\vec{q}$ (line 5). Then SAT^{sa} invokes a procedure called *Prop* to check Proposition 6 where C is used as the primary clause (line 6).

$SAT^{sa}(F, P, \vec{q})\{$
1 while *True* {
2      $(C_{cnfl}, \vec{q}) := BCP(F \cup P, \vec{q})$
3   if $(C_{cnfl} \neq nil)$ return(nil, C_{cnfl}, nil)
4   if $(Satisf(F, \vec{q}))$ return$(nil, nil, \vec{q})$
5      $C := PickCls(F)$
6      $(U, B, \vec{s}) := Prop(F, P, C, \vec{q})$
7   if $(U \neq nil)$
8   if $(CheckInvoc(U, \vec{q}))$ {
9              $P := P \cup \{U\}$; continue}
10   else return(U, nil, nil)
11   if $(B \neq nil)$ return(nil, B, nil)
12   return$(nil, nil, \vec{s})\}\}$

Fig. 2: Top procedure

If *Prop* derives a clause U that is unit in subspace $\vec{q}$, it stops checking Proposition 6 and returns U. If the current invocation of SAT^{sa} is the last where U is still unit, SAT^{sa} adds U to the set of proof clauses P. Then it starts a new iteration of the loop to satisfy U by BCP (lines 8-9). Otherwise, SAT^{sa} terminates returning U to the previous invocation (line 10). Here SAT^{sa} mimics the behavior of a CDCL solver after a conflict. Namely, the backtracking of such a solver to the farthest decision level where the derived conflict clause is still unit. If *Prop* proves Proposition 6 true, F is unsatisfiable in subspace $\vec{q}$. In this case, *Prop* outputs a proof clause B falsified by $\vec{q}$ that SAT^{sa} returns (line 11). Otherwise, *Prop* produces a satisfying assignment $\vec{s}$ returned by SAT^{sa} where $\vec{q} \subseteq \vec{s}$ (line 12).

10.2 *Prop* procedure

$Prop(F, P, C, \vec{q})\{$
1   $Lits := FreeLits(C, \vec{q})$
2   $ClsSet := GenSet(F, Lits, \vec{q})$
3   for every $C' \in ClsSet$ {
4   for every $l' \in Lits$ {
5   if $(l' \notin C')$ continue
6           $(U, B, \vec{s}) := Vic(F, P, C', l', \vec{q})$
7   if $(U \neq nil)$ return(U, nil, nil)
8   if $(\vec{s} \neq nil)$ return$(nil, nil, \vec{s})$
9           $P := P \cup \{B\}$ }}
10 $B := FormCls(ClsSet, P, \vec{q})$
11 return$(nil, B, nil)\}$

Fig. 3: Checking Proposition 6

The pseudocode of *Prop* is given in Fig. 3. It checks if Proposition 6 holds in subspace $\vec{q}$ when $C \in F$ is used as the primary clause. *Prop* starts with finding the set *Lits* of literals of C not falsified by $\vec{q}$ (line 1). Then it builds the set *ClsSet* of clauses of F unsatisfied by $\vec{q}$ that contain at least one literal of *Lits* i.e., the primary and secondary clauses (line 2). Note that since *ClsSet* is a subset of the formula F, it stays the same *regardless* of how many proof clauses are generated by SAT^{sa} (because they are *not* added to F).

Then *Prop* starts two nested *for* loops that check if Proposition 6 holds for F in subspace $\vec{q}$. The outer loop (lines 3-9) enumerates the clauses of *ClsSet*. The inner loop (lines 4-9) iterates over the literals of *Lits*. If a literal l' of *Lits* is present in $C' \in ClsSet$, *Prop* calls the procedure named *Vic* that checks the l'-vicinity of C' in subspace $\vec{q}$ (line 6).

The pseudocode of *Vic* is shown in Fig. 4. (First, *Vic* extends $\vec{q}$ to $\vec{q}'$ by adding the assignment specifying the l'-vicinity of C'. Then it calls SAT^{sa} to

check the satisfiability of F in subspace $\vec{q}\,'$.) Vic returns a clause U that is unit in subspace $\vec{q}$ or a clause B that is an l'-proof proof for C' in subspace $\vec{q}$ or a satisfying assignment $\vec{s}$. If Vic returns U, then $Prop$ terminates returning U (line 7). As we mentioned above, SAT^{sa} simulates here the behavior of a CDCL solver after a conflict. In this case, Proposition 6 is neither proved nor refuted in subspace $\vec{q}$. If Vic returns $\vec{s}$, then Proposition 6 does not hold in subspace $\vec{q}$ and $Prop$ terminates (line 8). If Vic returns an l'-proof clause B, $Prop$ adds it to the set P of proof clauses (line 9) and starts a new iteration.

$Vic(F,P,C',l',\vec{q})\{$
1 $\vec{q}\,':=\vec{q}\cup VicAssign(C',l')$
2 $(U,B,\vec{s}):=SAT^{sa}(F,P,\vec{q}\,')$
3 return$(U,B,\vec{s})$

Fig. 4: Checking vicinity

If the outer loop terminates, Proposition 6 holds. Then $Prop$ generates a proof clause B falsified by $\vec{q}$ and terminates (line 10-11). B consists of the literals falsified by the *relevant* variable assignments of $\vec{q}$. An example of the generation of B is given in Appendix E.

10.3   An example of how SAT^{sa} operates

Let us apply SAT^{sa} to the formula $F = C_1 \wedge \cdots \wedge C_7$ where $C_1 = x_1 \vee x_2$, $C_2 = \overline{x}_1 \vee x_3 \vee x_4$, $C_3 = x_2 \vee \overline{x}_3 \vee x_4$, $C_4 = x_2 \vee \overline{x}_4 \vee x_6$, $C_5 = x_2 \vee \overline{x}_4 \vee \overline{x}_6$, $C_6 = \overline{x}_2 \vee x_5$, $C_7 = \overline{x}_2 \vee \overline{x}_5$. Here we show a **fragment** of the operation of SAT^{sa}. (A full description is given in Appendix F.) Let SAT^{sa}_i,$Prop_i$ and $\vec{q}_i$ denote a call of SAT^{sa},$Prop$ and the current assignment $\vec{q}$ at recursion depth i.

The initial call is specified by SAT^{sa}_0 where $\vec{q}_0 = \emptyset$ and $P = \emptyset$. Assume that SAT^{sa}_0 picks $C_1 = x_1 \vee x_2$ and calls $Prop_0$ with C_1 as the primary clause. Then C_3, C_4, C_5 are the secondary clauses since they share the literal x_2 with C_1. To check Proposition 6, $Prop_0$ makes calls of SAT^{sa}_1 to examine the x_1-vicinity and x_2-vicinity of C_1 and the x_2-vicinity of C_3, C_4, C_5. Consider examining the x_1-vicinity of C_1. To this end, SAT^{sa}_1 is called in subspace $\vec{q}_1 = \{\boldsymbol{x_1 = 1}, x_2 = 0\}$. When SAT^{sa}_1 runs BCP in subspace $\vec{q}_1$, neither a conflict is encountered nor a satisfying assignment is produced.

Assume that then SAT^{sa}_1 picks $C_2 = \overline{x}_1 \vee x_3 \vee x_4$ and calls $Prop_1$ with C_2 as the primary clause. Then $C_3 = x_2 \vee \overline{x}_3 \vee x_4$ is the secondary clause since it shares the literal x_4 with C_2. At this point $Prop_1$ makes calls of SAT^{sa}_2 to examine the x_3-vicinity and x_4-vicinity of C_2 and the x_4-vicinity of C_3 in subspace $\vec{q}_1$. To examine the x_3-vicinity of C_2, SAT^{sa}_2 is called in subspace $\vec{q}_2 = \vec{q}_1 \cup \{\boldsymbol{x_3 = 1}, x_4 = 0\}$. Here $(\boldsymbol{x_3 = 1}, x_4 = 0)$ specifies the x_3-vicinity of C_2 in subspace $\vec{q}_1$. (Since $(C_2)_{\vec{q}_1} = \cancel{\overline{x}_1} \vee x_3 \vee x_4$.) When running BCP, SAT^{sa}_2 gets a conflict and produces a conflict clause $B_2 = \overline{x}_1 \vee x_2 \vee x_4$ obtained by resolving C_2 and C_3. The clause B_2 is an x_3-proof clause for C_2 in subspace $\vec{q}_1$ since $(B_2)_{\vec{q}_1} = \cancel{\overline{x}_1} \vee \cancel{x_2} \vee x_4$. So, $Prop_1$ adds B_2 to P. Then a new call of SAT^{sa}_2 is made to examine the x_4-vicinity of C_2 in subspace $\vec{q}_1$ and so on.

11   A Structure-Aware PQE Procedure

In this section, we sketch a structure-aware PQE solver called PQE^{sa} that does not add new quantified clauses (see Section 7). As in the case of SAT^{sa}, our

objective here is to present a powerful *direction* for further research. We assume that PQE^{sa} serves as a local plug-in solver for $EG\text{-}PQE^+$, a generic PQE algorithm [2] that we recall below. We also assume that $EG\text{-}PQE^+$ is used to take a clause C out of $\exists X[F(X,Y)]$. (One can take a *subset G* out of $\exists X[F]$ by using $EG\text{-}PQE^+$ to process the clauses of G one by one.)

11.1 Recalling $EG\text{-}PQE^+$

The pseudocode of $EG\text{-}PQE^+$ is shown in Fig. 5. It starts with initializing formulas $H_s(Y)$ and $H_u(Y)$ specifying subspaces $\vec{y}$ already examined. Here $\vec{y}$ is a full assignment to Y. If $\vec{y}$ falsifies H_s (or H_u), $F_{\vec{y}}$ is **s**atisfiable (respectively **u**nsatisfiable). Then $EG\text{-}PQE^+$ runs a *while* loop (lines 2-8). $EG\text{-}PQE^+$ aims to avoid the subspaces $\vec{y}$ where C is implied by $F \setminus \{C\}$ and hence trivially redundant. So, $EG\text{-}PQE^+$ checks if there is a full assignment $(\vec{y},\vec{x})$ to $Y \cup X$ falsifying C and satisfying $F\setminus\{C\}$ such that the subspace $\vec{y}$ was not examined yet (lines 3-4). (The existence of $(\vec{y},\vec{x})$ entails $(F \setminus \{C\})_{\vec{y}} \not\Rightarrow C_{\vec{y}}$.) If not, H_u is a solution i.e., $\exists X[F] \equiv H_u \wedge \exists X[F \setminus \{C\}]$. So, $EG\text{-}PQE^+$ terminates (line 5).

<div>

$EG\text{-}PQE^+(F,X,Y,C)$ {
1   $H_s := \emptyset;\ H_u := \emptyset$
2   while (*true*) {
3     $F' := (F \setminus \{C\}) \wedge \overline{C}$
4     $(\vec{y},\vec{x}) := Sat(H_s \wedge H_u \wedge F')$
5   if $((\vec{y},\vec{x}) = nil)$ return(H_u)
6     $(B,D) := PQE_{plg}(F,C,\vec{y})$
7   if $(B \neq nil)$ $H_u := H_u \cup \{B\}$
8   else $H_s := H_s \cup \{Cls(D)\}$}}

Fig. 5: $EG\text{-}PQE^+$

</div>

If $(\vec{y},\vec{x})$ exists, $EG\text{-}PQE^+$ calls PQE_{plg}, a plug-in PQE solver taking C out of $\exists X[F_{\vec{y}}]$ (line 6). If C is *not* redundant, PQE_{plg} returns a clause $B(Y)$ falsified by $\vec{y}$ as a solution to the PQE problem of line 6. That is $\exists X[F_{\vec{y}}] \equiv B \wedge \exists X[(F \setminus \{C\})_{\vec{y}}]$. The clause B is added to H_{uns} (line 7). If C *is* redundant in $\exists X[F_{\vec{y}}]$, PQE_{plg} returns a D-sequent D equal to $\vec{y}' \rightarrow C$ (where $\vec{y}' \subseteq \vec{y}$) stating redundancy of C in subspace $\vec{y}'$. $EG\text{-}PQE^+$ adds to H_{sat} the shortest clause falsified by $\vec{y}'$ to avoid entering the subspace $\vec{y}'$ again (line 8). In the context of $EG\text{-}PQE^+$, PQE_{plg} essentially works **as a SAT-solver** employing redundancy based reasoning. If C is redundant, $F_{\vec{y}}$ is *satisfiable* because $(\vec{y},\vec{x})$ satisfies $F \setminus \{C\}$. Otherwise, i.e., if C is *not* redundant, $F_{\vec{y}}$ is *unsatisfiable*.

11.2 Main idea behind PQE^{sa}

In this subsection, we give the main idea behind PQE^{sa} assuming that it is used as a plug-in solver for $EG\text{-}PQE^+$. (The pseudocode of PQE^{sa} is given in Appendix H.) Similarly to SAT^{sa}, PQE^{sa} is based on Proposition 6 and hence keeps proof clauses *separately* from the formula F. So, PQE^{sa} **does not add** new quantified clauses to the formula. Like SAT^{sa}, PQE^{sa} recursively calls itself. A call $PQE^{sa}(F,P,C,\vec{q})$ accepts the formula F, the current set of proof clauses P, a clause $C \in F$ and the current assignment $\vec{q}$. In the original call, $P = \emptyset$ and $\vec{q} = \vec{y}$ where $\vec{y}$ is computed by $EG\text{-}PQE^+$ (line 4 of Fig 5).

Similarly to SAT^{sa}, PQE^{sa} checks if Proposition 6 holds in subspace $\vec{q}$ when C is used as the primary clause. However, to perform this check, PQE^{sa} calls a procedure *Prop*∗ instead of *Prop*. The **main difference** of *Prop*∗ from *Prop* is

that the former calls a procedure named $Vic*$ (rather than Vic) that explores literal vicinities **approximately**. Namely, if $Vic*$ examines the l'-vicinity of a clause C' and fails to produce an l'-proof clause, it returns a D-sequent stating the *redundancy* of C' in subspace $\vec{q}$ rather than a *satisfying assignment*. So, if $Prop*$ fails to prove Proposition 6 with C as the primary clause, it returns a certificate of redundancy of C in subspace $\vec{q}$. For that reason, if $Prop*$ is invoked in the *initial* call $PQE^{sa}(F, P, C, \emptyset)$ and fails to prove Proposition 6 for the primary clause C, PQE^{sa} returns a D-sequent stating the redundancy of C in the subspace $\vec{y}$. As we mentioned above, this means that $F_{\vec{y}}$ is satisfiable. So, to check the satisfiability of $F_{\vec{y}}$, it suffices to explore literal vicinities *approximately*.

11.3 Procedures *Prop*∗ and *Vic*∗

$Prop*(F, P, C, \vec{q})$
1 for each second. clause C'
2   for each shared free lit. l'
3   check l'-vic. of C' by $Vic*$
4   if C' is redund.
5   store D-seq. ; remove C'
6   else add l'-proof clause to P
  - - - - -
7 for each free lit. l of C
8   check l-vic. of C by $Vic*$
9   if C is red. return D-seq.
10 else add l-proof clause to P
11 form proof clause; return it

Fig. 6: *Prop*∗

A description of $Prop*$ checking Proposition 6 in subspace $\vec{q}$ for the primary clause C is given in Fig. 6. (The pseudocode of $Prop*$ and $Vic*$ is given in Appendix H.) First, $Prop*$ processes secondary clauses (lines 1-6). For every secondary clause C' and for every unassigned literal l' shared by C' and C, the procedure $Vic*$ is called to check the l'-vicinity of C' in subspace $\vec{q}$. $Vic*$ either returns a D-sequent stating that C' is redundant in subspace $\vec{q}$, or an l'-proof clause. In C' is redundant, it is (temporarily) removed from F in subspace $\vec{q}$. Otherwise, the l'-proof clause is added to P.

Then $Prop*$ processes every unassigned literal l of the primary clause C (lines 7-11) calling $Vic*$ to check the l-vicinity of C in subspace $\vec{q}$. For the sake of simplicity, we **omit** the case when $Vic*$ produces a clause U that is unit in subspace $\vec{q}$. (This case works exactly as in SAT^{sa}.) So, in our description, $Vic*$ returns either a a D-sequent stating that C is redundant in subspace $\vec{q}$ (line 9) or an l-proof clause (line 10) . The former means that $Prop*$ failed to prove Proposition 6. If $Prop*$ reaches line 11, Proposition 6 holds. Then $Prop*$ uses the proof clauses and D-sequents stored earlier to produce a clause falsified by $\vec{q}$. (Appendix E shows how this clause is produced.)

A description of *Vic*∗ checking the l-vicinity of C in subspace $\vec{q}$ *approximately* is sketched in Fig. 7. First, $Vic*$ computes the assignment $\vec{q}'$ obtained from $\vec{q}$ by adding the assignment satisfying l and falsifying the other free literals of C (line 1). In SAT^{sa}, to do the check above, the Vic procedure calls a new invocation of SAT^{sa} to find out if F is *satisfiable* in subspace $\vec{q}'$. **In contrast to *Vic*,** $Vic*$ avoids generating a satisfying assignment. Instead, $Vic*$ checks the redundancy of each clause $C' \in F$ that contains $\bar{l}$ and is not satisfied by $\vec{q}'$ (lines 2-8). If every such a clause is proved redundant in subspace $\vec{q}'$, then C is **blocked** in subspace $\vec{q}$ (line 9). In this case, $Vic*$ generates a D-sequent stating redundancy of C in subspace $\vec{q}$ as described in [12].

```
Vic*(F, P, C, l, \vec{q}){
1  \vec{q}' := \vec{q} ∪ {l-vicin. of C }
2  G := clauses with \bar{l} and
3        unsatisf. by \vec{q}'
4  for each C' ∈ G {
5     run PQE^{sa} with C'
6        as the primary clause
7     if C' is redund. continue
8     else return l-proof clause }
9  C is blocked; return D-seq. }
```

Fig. 7: Checking l-vicinity

The redundancy of C' is tested by recursively calling PQE^{sa} (lines 5-6). It checks if Proposition 6 holds in subspace $\vec{q}'$ when C' is the primary clause. If not, C' is proved redundant and removed from F in subspace $\vec{q}'$ (line 7). Otherwise, a clause falsified by $\vec{q}'$ is produced that is an l-proof clause for C (line 8).

12 Conclusions

Virtually all efficient algorithms of hardware verification use some form of structure-aware computing (SAC). We relate SAC to Partial Quantifier Elimination (PQE). Interpolation, an instance of SAC, is as a special case of PQE. We show that SAC by PQE enables powerful methods of equivalence checking, model checking and testing via property generation. We also show that PQE solving itself benefit from being structure-aware. We formulate a new SAT algorithm based on SAC and use it to introduce a structure-aware PQE solver. Our discussion and results suggest that studying PQE and designing fast PQE solvers is of great importance.

References

1. E. Goldberg and P. Manolios, "Partial quantifier elimination," in *Proc. of HVC-14*. Springer-Verlag, 2014, pp. 148–164.
2. E. Goldberg, "Partial quantifier elimination and property generation," in *Enea, C., Lal, A. (eds) Computer Aided Verification, CAV-23,Lecture Notes in Computer Science, Part II*, vol. 13965, 2023, pp. 110–131.
3. ——, "Equivalence checking by logic relaxation," in *FMCAD-16*, 2016, pp. 49–56.
4. O. Kullmann, "New methods for 3-sat decision and worst-case analysis," *Theor. Comput. Sci.*, vol. 223, no. 1-2, pp. 1–72, 1999.
5. K. McMillan, "Applying sat methods in unbounded symbolic model checking," in *Proc. of CAV-02*. Springer-Verlag, 2002, pp. 250–264.
6. J. Jiang, "Quantifier elimination via functional composition," in *Proceedings of the 21st International Conference on Computer Aided Verification*, ser. CAV-09, 2009, pp. 383–397.
7. J. Brauer, A. King, and J. Kriener, "Existential quantification as incremental sat," ser. CAV-11, 2011, pp. 191–207.
8. W. Klieber, M. Janota, J.Marques-Silva, and E. Clarke, "Solving qbf with free variables," in *CP*, 2013, pp. 415–431.
9. N. Bjorner and M. Janota, "Playing with quantified satisfaction," in *LPAR*, 2015.
10. M. Rabe, "Incremental determinization for quantifier elimination and functional synthesis," in *CAV*, 2019.
11. E. Goldberg and P. Manolios, "Quantifier elimination by dependency sequents," in *FMCAD-12*, 2012, pp. 34–44.
12. ——, "Quantifier elimination via clause redundancy," in *FMCAD-13*, 2013, pp. 85–92.

13. E. Goldberg, "Partial quantifier elimination by certificate clauses," Tech. Rep. arXiv:2003.09667 [cs.LO], 2020.
14. The source of *ds-pqe*, http://eigold.tripod.com/software/ds-pqe.tar.gz.
15. The source of *EG-PQE$^+$*, http://eigold.tripod.com/software/eg-pqe-pl.1.0.tar.gz.
16. J. Marques-Silva and K. Sakallah, "Grasp – a new search algorithm for satisfiability," in *ICCAD-96*, 1996, pp. 220–227.
17. W. Craig, "Three uses of the herbrand-gentzen theorem in relating model theory and proof theory," *The Journal of Symbolic Logic*, vol. 22, no. 3, pp. 269–285, 1957.
18. K. McMillan, "Interpolation and sat-based model checking," in *CAV-03*. Springer, 2003, pp. 1–13.
19. G. Tseitin, "On the complexity of derivation in the propositional calculus," *Zapiski nauchnykh seminarov LOMI*, vol. 8, pp. 234–259, 1968, english translation of this volume: Consultants Bureau, N.Y., 1970, pp. 115–125.
20. A. Kuehlmann and F. Krohm, "Equivalence Checking Using Cuts And Heaps," *DAC*, pp. 263–268, 1997.
21. E. Goldberg, M. Prasad, and R. Brayton, "Using SAT for combinational equivalence checking," in *DATE-01*, 2001, pp. 114–121.
22. A. Mishchenko, S. Chatterjee, R. Brayton, and N. Een, "Improvements to combinational equivalence checking," in *ICCAD-06*, 2006, pp. 836–843.
23. B. L. Synthesis and V. Group, "ABC: A system for sequential synthesis and verification," 2017, http://www.eecs.berkeley.edu/∼alanmi/abc.
24. A. Biere, A. Cimatti, E. Clarke, M. Fujita, and Y. Zhu, "Symbolic model checking using sat procedures instead of bdds," in *DAC*, 1999, pp. 317–320.
25. E. Clarke, O. Grumberg, and D. Peled, *Model checking*. Cambridge, MA, USA: MIT Press, 1999.
26. K. McMillan, *Symbolic Model Checking*. Norwell, MA, USA: Kluwer Academic Publishers, 1993.
27. D. Kroening and O.Strichman, "Efficient computation of recurrence diameters," in *Proceedings of the 4th International Conference on Verification, Model Checking, and Abstract Interpretation*, 2002, p. 298–309.
28. M. Davis, G. Logemann, and D. Loveland, "A machine program for theorem proving," *Communications of the ACM*, vol. 5, no. 7, pp. 394–397, July 1962.
29. M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: engineering an efficient sat solver," in *DAC-01*, New York, NY, USA, 2001, pp. 530–535.
30. N. Eén and N. Sörensson, "An extensible sat-solver," in *SAT*, Santa Margherita Ligure, Italy, 2003, pp. 502–518.
31. A. Biere, "Picosat essentials," *JSAT*, vol. 4, no. 2-4, pp. 75–97, 2008.
32. E. Yolcu, "Lower bounds for set-blocked clauses proofs," Tech. Rep. arXiv:2401.11266 [cs.LO], 2024.
33. E. Goldberg, "Proving unsatisfiability of cnfs locally," *J. Autom. Reason.*, vol. 28, no. 4, pp. 417–434, 2002.
34. M. Abramovici, M. Breuer, and A. Friedman, *Digital Systems Testing and Testable Design*. John Wiley & Sons, 1994.
35. N. Bombieri, F. Fummi, and G. Pravadelli, "Hardware design and simulation for verification," in *6th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2006*, vol. 3965, pp. 1–29.
36. W. Lam, *Hardware Design Verification: Simulation and Formal Method-Based Approaches*. USA: Prentice Hall PTR, 2008.
37. P. Molitor, J. Mohnke, B. Becker, and C. Scholl, *Equivalence Checking of Digital Circuits Fundamentals, Principles, Methods*. Springer New York, NY, 2004.

38. C. Berman and L. Trevillyan, "Functional comparison of logic designs for vlsi circuits," pp. 456–459, 1989.
39. D. Brand, "Verification of large synthesized designs," pp. 534–537, 1993.
40. A. R. Bradley, "Sat-based model checking without unrolling," in *VMCAI*, 2011, pp. 70–87.

Appendix

A   Some Examples Of Structure-Aware Computing

As mentioned in the introduction, virtually all successful techniques of hardware verification use some form of structure-aware computing. In this appendix, we list some examples of that.

A.1   Testing

Testing is a ubiquitous technique of hardware verification [34,35,36]. One of the reasons for such omnipresence is that testing is surprisingly effective taking into account that only a minuscule part of the truth table is sampled. This effectiveness can be explained by the fact that current testing procedures check a *particular implementation* rather than sample the truth table. This is achieved by using some coverage metric that directs test generation at invoking a certain set of events. So, testing can be viewed as an instance of *structure-aware computing*.

A.2   Equivalence checking

Equivalence checking is one of the most efficient techniques of formal hardware verification [37]. Let N' and N'' be the circuits to check for equivalence. In general, equivalence checking is a hard problem that does not scale well even for combinational circuits. Fortunately, in practice, N' and N'' are *structurally similar*. In this case, one can often prove the equivalence of N' and N'' quite efficiently by using structure-aware computing. The latter is to locate internal points of N' and N'' linked by simple relations like equivalence [38,39,20,21,22]. The computation of these relations moves from inputs to outputs until the equivalence of the corresponding output variables of N' and N'' is proved. (If N' and N'' are sequential circuits, relations between internal points are propagated over multiple time frames.)

A.3   Model checking

A significant boost in hardware model checking has been achieved due to the appearance of IC3 [40]. The idea of IC3 is as follows. Let N be a sequential circuit. Let $P(S)$ be an invariant to prove where S is the set of state variables of N. (Proving P means showing that it holds in every reachable state of N.) IC3 looks for an *inductive* invariant P' such that $I \Rightarrow P' \Rightarrow P$ where $I(S)$ specifies the initial states of N. IC3 builds P' by constraining P via adding so-called

inductive clauses. The high scalability of IC3 can be attributed to the fact that in many cases P is *"almost" inductive*. So, to turn P into P', it suffices to add a relatively small number of inductive clauses. These clauses are specific to N i.e., to a particular implementation. So, building P' as a variation of P can be viewed as a form of *structure-aware computing*.

A.4 SAT solving

The success of modern SAT-solvers can be attributed to two techniques. The first technique is the conflict analysis introduced by GRASP [16]. The idea is that when a conflict occurs in the current subspace, one identifies the set of clauses responsible for this conflict. This set is used to generate a so-called conflict clause that is falsified in the current subspace. So, adding it to the formula diverts the SAT solver from any subspace where the same conflict occurs. Finding clauses involved in a conflict can be viewed as a form of *structure-aware computing*.

The second key technique of SAT solving introduced by Chaff [29] is to employ decision making that involves variables of recent conflict clauses. The reason why Chaff-like decision making works so well can be explained as follows. Assume that a SAT-solver with conflict clause learning checks the satisfiability of a formula F. Assume that F is unsatisfiable. (If F is satisfiable, the reasoning below can be applied to every subspace visited by this SAT-solver where F was unsatisfiable.) Learning and adding conflict clauses produces an unsatisfiable core of F that includes learned clauses. This core gradually shrinks in size and eventually reduces to an empty clause. The decision making of Chaff helps to identify the subset of clauses/variables of F making up the ever-changing unsatisfiable core that incorporates conflict clauses. So, one can also view the second technique as a form of *structure-aware computing*.

B   Proofs Of Propositions

B.1   Proofs of Section 2

Proposition 1. *Let a clause C be blocked in a formula $F(X,Y)$ at a variable $x \in X$. Then C is redundant in $\exists X[F]$ i.e., $\exists X[F \setminus \{C\}] \equiv \exists X[F]$.*

Proof. It was shown in [4] that adding a clause $B(X)$ blocked in $G(X)$ to the formula $\exists X[G]$ does not change the value of this formula. This entails that removing a clause $B(X)$ blocked in $G(X)$ does not change the value of $\exists X[G]$ either. So, B is redundant in $\exists X[G]$.

Let $\vec{y}$ be a full assignment to Y. Then the clause C of the proposition at hand is either satisfied by $\vec{y}$ or $C_{\vec{y}}$ is blocked in $F_{\vec{y}}$ at x. (The latter follows from the definition of a blocked clause.) In either case, $C_{\vec{y}}$ is redundant in $\exists X[F_{\vec{y}}]$. Since this redundancy holds in every subspace $\vec{y}$, C is redundant in $\exists X[F]$.

B.2 Proofs of Section 3

Proposition 2. *Formula $H(Y)$ is a solution to the PQE problem of taking G out of $\exists X[F(X,Y)]$ (i.e., $\exists X[F] \equiv H \wedge \exists X[F \setminus G]$) iff*

1. $F \Rightarrow H$ *and*
2. $H \wedge \exists X[F] \equiv H \wedge \exists X[F \setminus G]$

Proof. **The if part.** Assume that conditions 1, 2 hold. Let us show that $\exists X[F] \equiv H \wedge \exists X[F \setminus G]$. Assume the contrary i.e., there is a full assignment $\vec{y}$ to Y such that $\exists X[F] \neq H \wedge \exists X[F \setminus G]$ in subspace $\vec{y}$.

There are two cases to consider here. First, assume that F is satisfiable and $H \wedge \exists X[F \setminus G]$ is unsatisfiable in subspace $\vec{y}$. Then there is an assignment $(\vec{x}, \vec{y})$ satisfying F (and hence satisfying $F \setminus G$). This means that $(\vec{x}, \vec{y})$ falsifies H and hence F does not imply H. So, we have a contradiction. Second, assume that F is unsatisfiable and $H \wedge \exists X[F \setminus G]$ is satisfiable in subspace $\vec{y}$. Then $H \wedge F$ is unsatisfiable in subspace $\vec{y}$ too. So, condition 2 does not hold and we again have a contradiction.

The only if part. Assume that $\exists X[F] \equiv H \wedge \exists X[F \setminus G]$. Let us show that conditions 1 and 2 hold. Assume that condition 1 fails i.e., $F \not\Rightarrow H$. Then there is an assignment $(\vec{x}, \vec{y})$ satisfying F and falsifying H. This means that $\exists X[F] \neq H \wedge \exists X[F \setminus G]$ in subspace $\vec{y}$ and we have a contradiction. To prove that condition 2 holds, one can simply conjoin both sides of the equality $\exists X[F] \equiv H \wedge \exists X[F \setminus G]$ with H.

B.3 Proofs of Section 5

Proposition 3. *Assume M', M'' do not implement a constant (0 or 1). Let $\exists Z[Eq \wedge F] \equiv H(w', w'') \wedge \exists Z[F]$. Then M' and M'' are equivalent iff $H \Rightarrow (w' \equiv w'')$.*

Proof. **The if part.** Assume that $H \Rightarrow (w' \equiv w'')$. From Proposition 2 it follows that $(Eq \wedge F) \Rightarrow H$. So $(Eq \wedge F) \Rightarrow (w' \equiv w'')$. Recall that $F = F' \wedge F''$ where F' and F'' specify M' and M'' respectively. So, for every pair of inputs $\vec{v}'$ and $\vec{v}''$ satisfying $Eq(V', V'')$ (i.e., $\vec{v}' = \vec{v}''$), M' and M'' produce identical values of w' and w''. Hence, M' and M'' are equivalent.

The only if part. Assume the contrary i.e., M' and M'' are equivalent but $H(w', w'') \not\Rightarrow (w' \equiv w'')$. There are two possibilities here: $H(0,1) = 1$ or $H(1,0) = 1$. Consider, for instance, the first possibility i.e., $w' = 0, w'' = 1$. Since, M' and M'' are not constants, there is an input $\vec{v}'$ for which M' outputs 0 and an input $\vec{v}''$ for which M'' outputs 1. This means that the formula $H \wedge \exists Z[F]$ is satisfiable in the subspace $w' = 0, w'' = 1$. Then the formula $\exists Z[Eq \wedge F]$ is satisfiable in this subspace too. This means that there is an input $\vec{v}$ under which M' and M'' produce $w' = 0$ and $w'' = 1$. So, M' and M'' are inequivalent and we have a contradiction.

B.4 Proofs of Section 6

Proposition 4. *Let $k \geq 1$. Let $\exists S_{1,k}[I_1 \wedge I_2 \wedge T_{1,k}]$ be a formula where I_1 and I_2 specify the initial states of N in terms of variables of S_1 and S_2 respectively. Then $Diam(N, I) < k$ iff I_2 is redundant in $\exists S_{1,k}[I_1 \wedge I_2 \wedge T_{1,k}]$.*

Proof. **The if part.** Recall that $S_{1,k} = S_1 \cup \cdots \cup S_k$ and $T_{1,k} = T(S_1, S_2) \wedge \cdots \wedge T(S_k, S_{k+1})$. Assume that I_2 is redundant in $\exists S_{1,k}[I_1 \wedge I_2 \wedge T_{1,k}]$ i.e., $\exists S_{1,k}[I_1 \wedge T_{1,k}] \equiv \exists S_{1,k}[I_1 \wedge I_2 \wedge T_{1,k}]$. The formula $\exists S_{1,k}[I_1 \wedge T_{1,k}]$ is logically equivalent to R_k specifying the set of states of N reachable in k transitions. On the other hand, $\exists S_{1,k}[I_1 \wedge I_2 \wedge T_{1,k}]$ is logically equivalent to $\exists S_{2,k}[I_2 \wedge T_{2,k}]$ specifying the states reachable in $k-1$ transitions (because I_1 and $T(S_1, S_2)$ are redundant in $\exists S_{1,k}[I_1 \wedge I_2 \wedge T_{1,k}]$). So, redundancy of I_2 means that R_{k-1} and R_k are logically equivalent and hence, $Diam(N, I) < k$.

The only if part. Assume the contrary i.e., $Diam(N, I) < k$ but I_2 is *not* redundant in $\exists S_{1,k}[I_1 \wedge I_2 \wedge T_{1,k}]$. Then there is an assignment $\vec{p} = (\vec{s}_1, \ldots, \vec{s}_{k+1})$ such that a) $\vec{p}$ satisfies $I_1 \wedge T_{1,k}$; b) formula $I_1 \wedge I_2 \wedge T_{1,k}$ is unsatisfiable in subspace $\vec{s}_{k+1}$. (So, I_2 is not redundant because removing it from $I_1 \wedge I_2 \wedge T_{1,k}$ makes the latter satisfiable in subspace $\vec{s}_{k+1}$.) Condition b) means that $\vec{s}_{k+1}$ is unreachable in $k-1$ transitions whereas condition a) implies that $\vec{s}_{k+1}$ is reachable in k transitions. Hence $Diam(N, I) \geq k$ and we have a contradiction.

B.5 Proofs of Section 9

Proposition 5. *Let C be a clause of a formula $F(X)$. If F is satisfiable, there exists an assignment $\vec{x}$ satisfying F that a) satisfies only one literal of C or b) satisfies only one literal l of another clause $C' \in F$ where l is present in C.*

Proof. Let $\vec{p}$ be an assignment satisfying F. Let $Lits(C)$ denote the set of literals present in the clause C. Assume that $\vec{p}$ satisfies only one literal of C or there is a clause $C' \in F$ that contains a literal $l \in Lits(C)$ and l is the only literal of C' satisfied by $\vec{p}$. Then $\vec{p}$ is the satisfying assignment $\vec{x}$ we look for.

Now assume that $\vec{p}$ satisfies more than one literal in C and there is no clause C' of F that contains a literal l of $Lits(C)$ and l is the only literal satisfied by $\vec{p}$. Denote by $SatLits(C, \vec{p})$ the literals of C satisfied by $\vec{p}$. (So, $SatLits(C, \vec{p}) \subseteq Lits(C)$.) Below, we show that by flipping values of $\vec{p}$ satisfying literals of $SatLits(C, \vec{p})$, one eventually obtains a required satisfying assignment.

Let $l(x)$ be a literal of $SatLits(C, \vec{p})$. Denote by Q the set of clauses of F having at least one literal from $SatLits(s, \vec{p})$. Note that by our assumption, every clause of Q has at least two literals satisfied by $\vec{p}$. Denote by $\vec{p}\,'$ the assignment obtained from $\vec{p}$ by flipping the value of x. Let B be a clause of F. Note that if $B \notin Q$, the number of literals satisfied by $\vec{p}\,'$ is either the same as for $\vec{p}$ or increases by 1. Only if $B \in Q$, the number of literals satisfied by $\vec{p}\,'$ may decrease in comparison to $\vec{p}$. In particular, the number of literals of the clause C satisfied by $\vec{p}\,'$ is decreased by 1.

Denote by $SatLits(C, \vec{p}\,')$ the set of literals of C satisfied by $\vec{p}\,'$ (that is equal to $SatLits(C, \vec{p}) \setminus \{l(x)\}$). If there is a clause of Q for which $\vec{p}\,'$ satisfies only

one literal than it is a required satisfying assignment. If not, we remove from Q the clauses that do not contain a literal of $SatLits(C, \vec{p}\,')$ and continue the procedure above. Note that the updated set Q and $\vec{p}\,'$ preserve the property of Q and $\vec{p}$. That is $\vec{p}\,'$ satisfies at least two literals of every clause of Q. Now we pick another literal $l'(x')$ from $SatLits(C, \vec{p}\,')$ and flip the value of x'. Eventually such a procedure will produce an assignment that satisfies only one literal of either the clause C itself or some other clause of Q.

Proposition 6. *Let S be a formula falsified by every assignment meeting the condition a) or b) of Proposition 5 and $F \Rightarrow S$. Then F is unsatisfiable.*

Proof. Assume the contrary i.e., there is an assignment satisfying F. Then Proposition 5 entails that there exists an assignment $\vec{x}$ satisfying F that meets the condition a) and/or b). Hence, $\vec{x}$ falsifies S. Since $\vec{x}$ satisfies F, the latter cannot imply S and we have a contradiction.

C   A Single-Test Property

In this appendix, we formally define a single-test property. Let $M(X, V, W)$ be a combinational circuit where X, V, W denote the internal, input and output variables of M. Let $\vec{v}$ be a test and $\vec{w}$ be the output of M under $\vec{v}$. (Here $\vec{v}$ is a full assignment to the set V of *input* variables and $\vec{w}$ is a full assignment to the set W of *output* variables.) Let $\boldsymbol{H^{\vec{v}}(V, W)}$ be a formula such that $H^{\vec{v}}(\vec{v}\,', \vec{w}\,') = 1$ iff $\vec{v}\,' \neq \vec{v}$ or $(\vec{v}\,' = \vec{v}) \wedge (\vec{w}\,' = \vec{w})$. One can view $H^{\vec{v}}$ as describing the input/output behavior of M under the test $\vec{v}$. Let $F(X, V, W)$ be a formula specifying the circuit M as explained in Subsection 5.1. The formula $H^{\vec{v}}$ is implied by F and so it is a *property* of M. In Subsection 5.2, we refer to $H^{\vec{v}}$ as a **single-test property**.

One can obtain $H^{\vec{v}}$ by combining PQE with clause splitting. Let $V = \{v_1, \ldots, v_k\}$ and C be a clause of F. Let F' denote the formula obtained from F by replacing C with the following $k + 1$ clauses: $C_1 = C \vee \overline{l(v_1)}, \ldots,$

$C_k = C \vee \overline{l(v_k)}$, $C_{k+1} = C \vee l(v_1) \vee \cdots \vee l(v_k)$, where $l(v_i)$ is a literal of v_i. (So, $F' \equiv F$.) Consider the problem of taking the clause C_{k+1} out of $\exists X[F']$. One can show [2] that a) this PQE problem has *linear* complexity; b) taking out C_{k+1} produces a *single-test property* $H^{\vec{v}}$ corresponding to the test $\vec{v}$ falsifying the literals $l(v_1), \ldots, l(v_k)$.

D   Computing Rel_i By CP^{pqe}

In Subsection 5.3, we recalled CP^{pqe}, a method of equivalence checking by PQE introduced in [3]. In this appendix, we describe how CP^{pqe} computes the formula Rel_i specifying relations between cut points of Cut_i. We reuse the notation of Subsection 5.3.
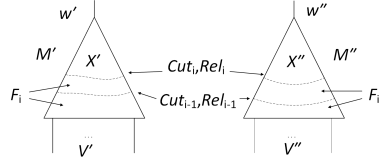
Fig. 8: Computing Rel_i in CP^{pqe}

Let formula F_i specify the gates of M' and M'' located between their inputs and Cut_i (see Fig. 8). Let Z_i denote the variables of F_i minus those of Cut_i. Then Rel_i is obtained by taking Rel_{i-1} out of $\exists Z_i[Rel_{i-1} \wedge F_i]$ i.e., $\exists Z_i[Rel_{i-1} \wedge F_i] \equiv Rel_i \wedge \exists Z_i[F_i]$. The formula Rel_i depends only on variables of Cut_i. (The other variables of $Rel_{i-1} \wedge F_i$ are in Z_i and hence, quantified.) Since Rel_i is obtained by taking out Rel_{i-1}, the latter is redundant in $\exists Z_i[Rel_{i-1} \wedge Rel_i \wedge F_i]$. One can show that this property *implies* the property mentioned in Subsection 5.3: Rel_{i-1} is redundant in $\exists Z[Rel_{i-1} \wedge Rel_i \wedge F]$. So, the latter is just a *weaker* version of the former.

E   Examples Of How A Proof Clause Is Generated

In this appendix, we explain how a proof clause is generated when Proposition 6 holds in subspace $\vec{q}$. Denote this clause as B. The main idea here is the same as in generation of a conflict clause by a CDCL solver. Namely, B consists of the literals falsified by the values of $\vec{q}$ that *mattered* in proving Proposition 6. In Example 5 we describe how B is generated by the *Prop* procedure of SAT^{sa}. Example 6 explains generation of B by *Prop∗* of PQE^{sa} described in Appendix H.2.

Example 5. Here we give an example of how the procedure *FormCls* of *Prop* (line 11 of Fig. 3) generates a proof clause. Let $F(X)$ be equal to $C_1 \wedge C_2 \wedge C_3 \wedge \ldots$ where $C_1 = x_1 \vee x_2$, $C_2 = x_1 \vee x_5$ and $C_3 = x_2 \vee \overline{x}_6 \vee x_8$. Let C_1, C_2, C_3 be the only clauses of F with literals x_1 and/or x_2 (but F can contain any number of clauses with $\overline{x}_1$ and/or $\overline{x}_2$). Assume that SAT^{sa} is applied to F and that the current assignment $\vec{q}$ equals $(x_3 = 0, x_4 = 1, x_5 = 1, x_7 = 0, x_9 = 0, x_{10} = 1)$. Assume also that *Prop* is called to check if Proposition 6 holds in subspace $\vec{q}$ when C_1 is used as the primary clause. (Then C_2, C_3 are the secondary clauses.)

   Assume that Proposition 6 is true and *Prop* generates proof clauses B_1, B_1' and B_3 for C_1 and C_3 in subspace $\vec{q}$. (*Prop* ignores the secondary clause C_2 since it is satisfied by $\vec{q}$.) Namely, $B_1 = x_3 \vee x_2$, $B_1' = \overline{x}_4 \vee x_1$, $B_3 = \overline{x}_6 \vee x_9$. Here, B_1 and B_1' are x_1-proof and x_2-proof clauses for C_1 in subspace $\vec{q}$ respectively. The clause B_3 is an x_2-proof clause for C_3 in subspace $\vec{q}$. Then *FormCls* will produce the clause $B = x_3 \vee \overline{x}_4 \vee x_9$ that includes the literals of the proof clauses B_1, B_1' and B_3 falsified by $\vec{q}$. Note that B does not contain the literal $\overline{x}_5$ falsified by the assignment $x_5 = 1$ of $\vec{q}$ satisfying C_2. The reason is that since B_1, B_1', B_3 do not depend on x_5, they could be derived even if x_5 was assigned 0. This implies that C_2 is redundant in any subspace falsifiying the clause B and so can be ignored.

Example 6. Now we consider generation of a proof clause by PQE^{sa}. Namely, we describe how the procedure *FormCls* of *Prop∗* (line 15 of Fig. 12) generates a

proof clause. The difference here is that some clauses relevant to proving Proposition 6 may be *proved* redundant. Then, when generating a proof clause, one needs to take into account the D-sequents of redundant clauses.

Let us consider a modification of Example 5 where one deals with the same formula F. As before we assume that C_1 is the primary clause and C_2, C_3 are the secondary clauses. Assume that the current assignment is *different* in the value of x_5 that is $\vec{q}$ equals $(x_3 = 0, x_4 = 1, \boldsymbol{x_5 = 0}, x_7 = 0, x_9 = 0, x_{10} = 1)$. So, the clause C_2 is *not satisfied* by $\vec{q}$. Assume that $Prop*$ generated the same proof clauses B_1, B_1', B_3 for C_1 and C_3 as in Example 5. Now assume that C_2 is *proved* redundant in subspace $\vec{q}$ and $(x_{10} = 1) \to C_2$ is the corresponding D-sequent.

Then *FormCls* produces the same proof clause $B = x_3 \vee \overline{x}_4 \vee x_9$ consisting of the literals of B_1, B_1', B_3 falsified by $\vec{q}$. Note that B does not contain the literal $\overline{x}_{10}$ reflecting the fact that C_2 is proved redundant in subspace $x_{10} = 1$. The reason is the same as in the previous example. That is, since B_1, B_1', B_3 do not depend on x_{10}, they could be derived even if x_{10} was assigned 0.

F   An Example Of How SAT^{sa} Operates

In this appendix, we show how SAT^{sa} operates on a small example. We partly described the operation of SAT^{sa} on this example in Subsection 10.3. That is we apply SAT^{sa} to the formula $F = C_1 \wedge \cdots \wedge C_7$ where $C_1 = x_1 \vee x_2$, $C_2 = \overline{x}_1 \vee x_3 \vee x_4$, $C_3 = x_2 \vee \overline{x}_3 \vee x_4$, $C_4 = x_2 \vee \overline{x}_4 \vee x_6$, $C_5 = x_2 \vee \overline{x}_4 \vee \overline{x}_6$, $C_6 = \overline{x}_2 \vee x_5$, $C_7 = \overline{x}_2 \vee \overline{x}_5$. The control flow of SAT^{sa} is shown in Fig. 9. Here $\vec{q}_i$ is the current assignment. The subscript in SAT_i^{sa}, $Prop_i$ and $\vec{q}_i$ gives the recursion depth. Nd_j is a node of the search tree.
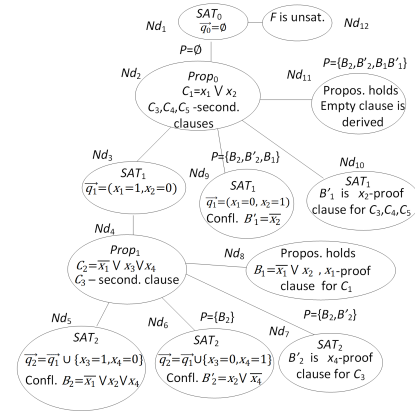

Fig. 9: Control flow

The call SAT_0^{sa} is specified by the node Nd_1. SAT_0^{sa} picks C_1 and calls $Prop_0$ with C_1 as the primary clause (node Nd_2). C_3, C_4, C_5 are the secondary clauses sharing the literal x_2 with C_1. To check Proposition 6, $Prop_0$ makes calls of SAT_1^{sa} to examine the x_1-vicinity and x_2-vicinity of C_1 (nodes Nd_3, Nd_9), and the x_2-vicinity of C_3, C_4, C_5 (node Nd_{10}). In the node Nd_3, SAT_1^{sa} is called in subspace $\vec{q}_1 = \{\boldsymbol{x_1 = 1}, x_2 = 0\}$ (the x_1-vicinity of C_1). SAT_1^{sa} picks C_2 and calls $Prop_1$ with C_2 as the primary clause. C_3 is the secondary clause sharing x_4 with C_2. $Prop_1$ makes calls of SAT_2^{sa} to examine the x_3-vicinity and x_4-vicinity of C_2 (nodes Nd_5, Nd_6), and the x_4-vicinity of C_3 in subspace $\vec{q}_1$ (node Nd_7).

In the node Nd_5, the subspace $\vec{q}_2 = \vec{q}_1 \cup \{\boldsymbol{x_3 = 1}, x_4 = 0\}$ is examined (the x_3-vicinity of C_2 in subspace $\vec{q}_1$). When running BCP, SAT_2^{sa} gets a conflict

and produces a conflict clause $B_2 = \overline{x}_1 \vee x_2 \vee x_4$ by resolving C_2 and C_3. The clause B_2 is an x_3-proof clause for C_2 in subspace $\vec{q}_1$ since $(B_2)_{\vec{q}_1} = \overline{\cancel{x}_1} \vee \cancel{x_2} \vee x_4$. So, $Prop_1$ adds B_2 to P before entering Nd_6.

In the node Nd_6, SAT_2^{sa} is called to explore the x_4-vicinity of C_2 in subspace $\vec{q}_1$. So, $\vec{q}_2 = \vec{q}_1 \cup \{x_3 = 0, \boldsymbol{x_4 = 1}\}$. Here SAT_2^{sa} gets a conflict and produces a conflict clause $B_2' = x_2 \vee \overline{x}_4$ by resolving C_4 and C_5. So, B_2' is an x_4-clause for C_2 in subspace $\vec{q}_1$ and $Prop_1$ adds B_2' to P. Since B_2' is also an x_4-proof clause for C_3 in subspace $\vec{q}_1$ (node Nd_7), Proposition 6 holds and F is unsatisfiable in subspace $\vec{q}_1$. $Prop_1$ generates the clause $B_1 = \overline{x}_1 \vee x_2$ falsified by $\vec{q}_1$ (node Nd_8). B_1 is an x_1-proof clause for C_1 whose computation started in Nd_3.

In the node Nd_9, SAT_1^{sa} gets a conflict producing the conflict clause $B_1' = \overline{x}_2$ obtained by resolving C_6 and C_7. B_1' is an x_2-proof clause for C_1. This clause is also an x_2-proof clause for C_3, C_4, C_5 (node Nd_{10}). So, Proposition 6 holds for $Prop_0$ in subspace $\vec{q}_0 = \emptyset$ (node Nd_2). So, the formula F is **unsatisfiable** (node Nd_{11}). At this point, the set P of proof clauses consists of B_2, B_2', B_1, B_1'.

G Tuning $\boldsymbol{SAT^{sa}}$ To A Class Of Formulas
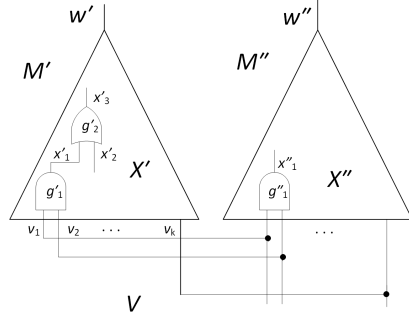

Fig. 10: Checking M' and M'' for equivalence

In this appendix, we describe how SAT^{sa} can be tuned to a particular application, namely, equivalence checking. Consider checking the equivalence of circuits M' and M'' shown in Fig. 10. This problem reduces to testing the satisfiability of the formula $F = F' \wedge F'' \wedge A_1 \wedge A_2$ where $F'(X', V, w')$, $F''(X'', V, w'')$ specify M' and M'' and $A_1 = w' \vee w''$, $A_2 = \overline{w}' \vee \overline{w}''$ specify $w' \neq w''$. Here X', X'' specify the sets of internal variables w', w'' denote the output variables of M' and M''. The set V specifies the input variables shared by M' and M''. In this appendix we reuse the notation of Appendix F. That is SAT_i^{sa} and $\vec{q}_i$ specify the invocation of SAT^{sa} and cur-

rent assignment $\vec{q}$ at recursion level i. (For the sake of simplicity, we omit mentioning the function $Prop_i$ called by SAT_i^{sa} to check Proposition 6.) The idea of tuning is to mimic the behavior of the specialized equivalence checker CP described in Subsection 5.3.

Recall that CP derives some relationships between cut points of M' and M'' moving from inputs to outputs. SAT^{sa} can simulate this behavior by processing the clauses of F from outputs to inputs. That is the current invocation SAT_i^{sa} looks for the next primary clause among the clauses that a) are not satisfied by $\vec{q}_i$ yet and b) specify gates that are the closest to the outputs. This guarantees that calls SAT_i^{sa} with the largest depth i produce proof clauses relating internal

points of M' and M'' that are the closest to inputs. By going from deeper to more shallow recursion depths, SAT^{sa} actually moves from inputs to outputs.

Let us consider the operation of SAT^{sa} in more detail. The clauses of F that are the "closest" to outputs are A_1 and A_2. So, the original invocation SAT_0^{sa} picks one of them as the primary clause. Assume that it is the clause $A_1 = w' \vee w''$. Suppose that SAT_0^{sa} derived an w'-proof clause e.g. $B_1 = \overline{w}' \vee w''$ and an w''-proof clause e.g. $B_1^* = w' \vee \overline{w}''$ for A_1. Then one can trivially derive proof clauses for the *secondary* clauses of F (i.e., those containing literals w' and w'') from $A_1 \wedge A_2 \wedge B_1 \wedge B_1^*$ because the latter is unsatisfiable.

Assume that SAT_0^{sa} explores the w'-vicinity of A_1 i.e., $(\boldsymbol{w' = 1}, w'' = 0)$. Assume that eventually SAT_i^{sa} is called after the gates g_2' and g_1'' of N' and N'' are reached (see Fig. 10). That is $\vec{q}_i$ contains assignments $x_3' = 1$ and $x_1'' = 0$ to the output variables of g_2' and g_1''. The formulas $F_{g_1'}$, $F_{g_2'}$, $F_{g_1''}$ specifying those gates and the gate g_1' are as follows:

$F_{g_1'} = C_1' \wedge C_2' \wedge C_3'$ where $C_1' = \overline{v}_1 \vee \overline{v}_2 \vee x_1'$, $C_2' = v_1 \vee \overline{x}_1'$, $C_3' = v_2 \vee \overline{x}_1'$,
$F_{g_2'} = C_4' \wedge C_5' \wedge C_6'$ where $C_4' = x_1' \vee x_2' \vee \overline{x}_3'$, $C_5' = \overline{x}_1' \vee x_3'$, $C_6' = \overline{x}_2' \vee x_3'$,
$F_{g_1''} = C_1'' \wedge C_2'' \wedge C_3''$ where $C_1'' = \overline{v}_1 \vee \overline{v}_2 \vee x_1''$, $C_2'' = v_1 \vee \overline{x}_1''$, $C_3'' = v_2 \vee \overline{x}_1''$.

Assume that SAT_i^{sa} picks $C_4' = x_1' \vee x_2' \vee \overline{x}_3'$ as the primary clause. Suppose that in contrast to x_3', the variables x_1' and x_2' are not assigned in $\vec{q}_i$. Assume that SAT_{i+1}^{sa} is invoked to explore the x_1'-vicinity of C_4' specified by $(\boldsymbol{x_1' = 1}, x_2' = 0, x_3' = 1)$. So, $\vec{q}_{i+1} = \vec{q}_i \cup \{x_1' = 1, x_2' = 0\}$. When running BCP, a conflict occurs in subspace $\vec{q}_{i+1}$ and the conflict clause $C_{cnfl} = \overline{x}_1' \vee x_1''$ is derived. (The latter is obtained by resolving C_1'' with C_2' and C_3'.) The clause C_{cnfl} is an x_1'-proof clause for C_4' in subspace $\vec{q}_i$ (where $x_1'' = 0$). So, C_{cnfl} is added to P.

Note that the proof clause C_{cnfl} above relates two internal variables of M' and M''. This clause can be used by calls SAT_j^{sa} where $j < i$ to derive proof clauses relating internal points located further away from inputs.

H Pseudocode of PQE^{sa}

$PQE^{sa}(F, P, C, \vec{q})\{$
1 $(C_{cnfl}, \vec{q}) := BCP(F \cup P, \vec{q})$
2 if $(C_{cnfl} \neq nil)$
3   return(C_{cnfl}, nil)
4 if $(Satisf(F, \vec{q}))$ {
5     $D := FormDseq(F, C, \vec{q})$
6   return(nil, D)}
7 $(B_{prf}, D) := Prop*(F, P, C, \vec{q})$
8 if $(B_{prf} \neq nil)$ {
9   return(B_{prf}, nil)}
10 return(nil, D)}

Fig. 11: Top procedure of PQE^{sa}

In this appendix, we provide the pseudocode of PQE^{sa} introduced in Section 11.

H.1 Top procedure of PQE^{sa}

In the rest of this appendix, we describe PQE^{sa} assuming that it is used as a plug-in solver for $EG\text{-}PQE^+$ (see Subsection 11.1). Similarly to SAT^{sa}, PQE^{sa} is based on Proposition 6 and hence keeps the original formula intact. That is, PQE^{sa} **does not add** new quantified clauses to the formula. Like SAT^{sa}, PQE^{sa} recursively calls itself.

The top procedure of PQE^{sa} (Fig. 11) is similar to that of SAT^{sa}. PQE^{sa} accepts the

formula F, the current set of proof clauses P, a clause C of F and the current assignment $\vec{q}$. PQE^{sa} returns a clause falsified by $\vec{q}$ (F is unsatisfiable in subspace $\vec{q}$) or a D-sequent stating the redundancy of C in subspace $\vec{q}$ (the satisfiability of F in subspace $\vec{q}$ is unknown). In the original call, $P = \emptyset$ and $\vec{q} = \vec{y}$ where $\vec{y}$ is a full assignment to Y (see Fig. 5, line 4).

Like SAT^{sa}, PQE^{sa} first runs BCP (line 1). In case of a conflict, a conflict clause C_{cnfl} is generated and PQE^{sa} terminates returning C_{cnfl} (lines 2-3). Although PQE^{sa}, in general, does not produce a satisfying assignment, BCP may accidentally find one (lines 4-6). In this case, PQE^{sa} produces a D-sequent stating the redundancy of C in subspace $\vec{q}$ as described in [12]. Then SAT^{sa} terminates returning D. If BCP does not encounter a terminating condition, PQE^{sa} calls a procedure named $Prop*$ to check if Proposition 6 holds where C is used as the primary clause (line 7). As we mentioned in Subsection 11.3, for the sake of simplicity, we *ignore* the case when the procedure $Vic*$ below (and hence $Prop*$) returns a clause U that is unit in subspace $\vec{q}$.

If Proposition 6 holds, then $Prop*$ returns a clause B_{prf} falsified by $\vec{q}$. The **difference** of $Prop*$ from $Prop$ is that the former does not return a satisfying assignment if it fails to prove Proposition 6. Instead, $Prop*$ returns a D-sequent D stating the redundancy of C in subspace $\vec{q}$. After calling $Prop*$, PQE^{sa} terminates returning B_{prf} or D (lines 8-10).

H.2 *Prop*∗* procedure

```
Prop*(F, P, C, q⃗){
1   Lits := FreeLits(C, q⃗)
2   ClsSet := GenSet(F, Lits, q⃗)
3   Ds := ∅
4   for every C′ ∈ ClsSet {
5     for every l′ ∈ Lits {
6       if (l′ ∉ C′) continue
7       (B_prf, D) := Vic*(F, P, C′, l′, q⃗)
8       if (D ≠ nil) {
9         if (C′ = C) goto Finish
10        RemCls(F, C′, q⃗)
11        Ds := Ds ∪ {D}}
12      if (Falsif(B_prf, q⃗))
13        goto Finish
14      P := P ∪ {B_prf}}}}
15  B_prf := FormCls(F, P, Ds, q⃗)
Finish:
16  RecCls(F, q⃗)
17  return(B_prf, D)}
```

Fig. 12: Checking Proposition 6

Assume that PQE^{sa} is used to take a clause C out of $\exists X[F]$ in subspace $\vec{q}$. $Prop*$ is called by PQE^{sa} to check if Proposition 6 holds in subspace $\vec{q}$ if C is used as the primary clause. The pseudocode of $Prop*$ is given in Fig. 12. $Prop*$ accepts the formula F, the current set P of proof clauses, a clause $C \in F$ and assignment $\vec{q}$. If $Prop*$ succeeds in proving Proposition 6, it returns a proof clause B_{prf} falsified by $\vec{q}$. Otherwise, it returns a D-sequent D stating the redundancy of the primary clause C in subspace $\vec{q}$.

$Prop*$ starts with finding the set $Lits$ of literals of C not falsified by $\vec{q}$ and building the set $ClsSet$ of clauses of F with at least one literal of $Lits$ that are not satisfied by $\vec{q}$ (lines 1-2). That is $ClsSet$ consists of the primary clause C and secondary clauses. Besides, $Prop*$ initializes the set Ds of D-sequents for the clauses of $ClsSet$ proved redundant

in subspace $\vec{q}$ (line 3). After that, *Prop*∗ starts two nested *for* loops that check if Proposition 6 holds for $\exists X[F]$ in subspace $\vec{q}$ (lines 4-14).

In each iteration of the outer loop, a clause $C' \in \mathit{ClsSet}$ is selected. (We assume here that the secondary clauses of *ClsSet* are processed *before* the primary clause C.) Then the inner loop enumerates every literal l' of *Lits*. If a literal l' of *Lits* is present in C', the procedure called *Vic*∗ is invoked to check the l'-vicinity of C' in subspace $\vec{q}$ (lines 6-7).

If *Vic*∗ does not produce an l'-proof clause P, it returns a D-sequent D stating that C' is redundant in subspace $\vec{q}$. Then *Prop*∗ does the following. If C' is the primary clause (i.e., $C' = C$), then *Prop*∗ terminates and returns D indicating that it *failed* to prove Proposition 6 (line 9). Otherwise, C' is temporarily removed from the formula as redundant (line 10). Besides, *Prop*∗ adds to Ds the D-sequent D (line 11).

In reality, when a secondary clause C' is proved redundant, one needs to check the validity of the proof clauses previously generated in this call of *Prop*∗. For the sake of simplicity, we do not mention this fact in Fig. 12. Instead, we discuss this issue in Subsection H.4.

If *Vic*∗ returns an l'-proof clause B_{prf} for C', then *Prop*∗ first checks if B_{prf} is falsified by the current assignment $\vec{q}$. If so, *Prop*∗ terminates returning B_{prf} (line 12-13). Otherwise, *Prop*∗ adds B_{prf} to the set of proof clauses P. If *Prop*∗ reaches the end of the outer loop, Proposition 6 holds. *Prop*∗ calls the procedure named *FormCls* to generate a clause B_{prf} falsified by $\vec{q}$ (line 15). Generation of such a clause is described in Appendix E.

Finally *Prop*∗ puts the secondary clauses proved redundant in subspace $\vec{q}$ back to the formula (line 16). Then it returns B_{prf} or the D sequent stating redundancy of the primary clause C in subspace $\vec{q}$ (line 17). At this point, one of them is *nil*.

$Vic*(F, P, C, l(x), \vec{q})\{$
1    $Ds := \emptyset$
2    $\vec{r} := \vec{q} \cup VicAssgn(C, l, \vec{q})$
3    $T := ResCls(F, C, l, \vec{r})$
4   for every clause $C' \in T$ {
5      $(B_{prf}, D) := PQE^{sa}(F, P, C', \vec{r})$
6   if $(B_{prf} = nil)$ {
7       $RemCls(F, C', \vec{r})$
8       $Ds := Ds \cup \{D\}$
9   continue }
10   else goto *Finish* }
11   $D := BlkdCls(F, T, Ds, \vec{q})$
Finish:
12   $RecCls(F, T, Ds, \vec{q})$
13   return$(B_{prf}, D)\}$

Fig. 13: Checking vicinity

H.3   *Vic*∗ procedure

The pseudocode of the *Vic*∗ procedure is shown in Fig. 13. It accepts the formula F, the current set P of proof clauses, a clause $C \in F$, a literal $l(x)$ of a variable x present in C and the current assignment $\vec{q}$. *Vic*∗ returns either a clause B_{prf} that is an l-proof for C in subspace $\vec{q}$ or a D-sequent D. The latter states that C is redundant in subspace $\vec{q}$ because it is blocked at x.

First, *Vic*∗ initializes the set Ds that accumulates the D-sequents of clauses proved redundant in subspace $\vec{q}$ (line 1). Then it generates the assignment $\vec{r}$ obtained by adding to $\vec{q}$ the assignment specifying the l-vicinity of C (line 2). After that, *Vic*∗ forms the set T of clauses with the literal $\bar{l}$ that are not satisfied by $\vec{r}$. These are the clauses resolvable with C on x in subspace $\vec{r}$.

Then $Vic*$ runs a *for* loop (lines 4-10). In each iteration of this loop, $Vic*$ checks the redundancy of a clause $C' \in T$ in subspace $\vec{r}$. This is done by calling PQE^{sa} in subspace $\vec{r}$ (line 5). PQE^{sa} either returns a clause B_{prf} falsified by $\vec{r}$ or produces a D-sequent D stating that C' is redundant in subspace $\vec{r}$. In the latter case, $Vic*$ temporarily removes C' from the formula and adds D to the set of D-sequents Ds (lines 6-9). If PQE^{sa} returns B_{prf}, the latter is the l-proof clause for C in subspace $\vec{q}$. So, $Vic*$ terminates (line 10).

If all clauses of T are proved redundant in subspace $\vec{r}$, then C is blocked at variable x in subspace $\vec{r}$. So, C is redundant in this subspace and $Vic*$ generates a D-sequent stating this fact (line 11). Generation of such a D-sequent is described in [12]. Finally, $Vic*$ recovers all clauses temporarily removed from the formula in the loop and terminates returning B_{prf} and D (lines 12-13). At this point one of them is *nil*.

H.4 Consistency of proof clauses and D-sequents

To produce correct results, PQE^{sa} needs to maintain some form of consistency between proof clauses and D-sequents. Suppose, for instance, that PQE^{sa} does computations in a subspace $\vec{q}$ and temporarily removes a clause $C \in F$ as proved redundant in this subspace. Then PQE^{sa} should not employ proof clauses derived using C. Assume, for instance, that a secondary clause C' is proved redundant and temporarily removed in a call of *Prop* (line 10 of Fig. 12). Then one should check the proof clauses previously added to P (line 14) and recompute those that were derived using C'. A straightforward way to achieve the consistency above is to a) store the list clauses used to derive each proof clause B_{prf} and b) avoid reusing B_{prf} if one of the clauses on this list is currently removed as redundant. However, more efficient methods can be designed that reduce the amount of information one needs to store.