Software for Quantifier Elimination in Propositional Logic

Eugene Goldberg[1] and Panagiotis Manolios[2]

[1] Northeastern University, USA
eigold@ccs.neu.edu,
http://www.ccs.neu.edu/home/eigold
[2] Northeastern University, USA
pete@ccs.neu.edu,
http://www.ccs.neu.edu/home/pete

Abstract. We consider the problem of Quantifier Elimination (QE): given a Boolean CNF formula F where some variables are existentially quantified, find a logically equivalent quantifier-free CNF formula. This problem can be solved by finding a set of clauses containing only free variables such that adding this set of clauses to F makes all of the clauses of F containing quantified variables redundant. To solve the QE problem we developed a tool that handles a more general problem called partial QE. Our tool generates a set of clauses that when added to F render a specified subset of clauses with quantified variables redundant. In particular, if the specified subset contains all the clauses with quantified variables, our tool performs QE.

Keywords: Propositional logic, quantifier elimination, dependency sequents

1 Introduction

In this extended abstract, we describe software for solving the problem of Quantifier Elimination (QE) and the Partial QE (PQE). Let $H(X, Y)$ be a Boolean formula in Conjunctive Normal Form (CNF). Given a formula $\exists X[H]$, the **QE problem** is to find a CNF formula $H^*(Y)$ such that $H^* \equiv \exists X[H]$.

Let $F(X, Y)$ and $G(X, Y)$ be CNF formulas. Given a formula $\exists X[F \wedge G]$, the **PQE problem** is to find a CNF formula $F^*(Y)$ such that $F^* \wedge \exists X[G] \equiv \exists X[F \wedge G]$. We will say that formula F^* is obtained by taking F out of the scope of quantifiers. Obviously, QE is a special case of PQE where the entire formula is taken out of the scope of quantifiers.

QE has numerous applications in verification. For instance, to find if a system specified by a transition relation $T(S, S')$ can reach a bad state, one needs to perform reachability analysis. Here S, S' specify current and next state variables. The set of states reachable in one transition from states specified by Boolean formula $G(S')$ is described by $\exists S[G \wedge T]$. To represent this set of states in a quantifier-free form one needs to find a quantifier-free formula logically equivalent to $\exists S[G \wedge T]$, *i.e.*, to solve the QE problem.

Unfortunately, the "straightforward" methods of QE seem to be very time-consuming even in propositional logic. This is one reason that many successful theorem proving methods such as interpolation [4] and IC3 [1] avoid QE and use SAT-based reasoning instead. This motivates our interest in studying variations of QE that can be solved efficiently. PQE is one such variation. A detailed description of our algorithm for solving the PQE problem is given in [3].

2 Application of PQE: Solving SAT by PQE

In [3], we list some applications of PQE to verification problems. In this section, we give one more application not mentioned in [3]. Namely, we show how PQE can be used to solve a version of SAT called Circuit-SAT. We give two methods of reducing Circuit-SAT to PQE that are complementary to each other.

2.1 Circuit-SAT

Let $N(X, Y, z)$ be a single-output combinational circuit, where X, Y, and z are input variables, internal variables, and the output of N, respectively. Suppose that one needs to check the satisfiability of N, *i.e.*, whether N ever evaluates to 1. We will refer to this problem as Circuit-SAT (in contrast with SAT, the problem of checking the satisfiability of arbitrary Boolean formulas). A common way of solving Circuit-SAT is to represent N as the CNF formula $H(X, Y, z)$ obtained by the Tseitsin transformation and to check if $H \wedge z$ is satisfiable.

2.2 Reducing Circuit-SAT to PQE: First Method

One can reduce checking the satisfiability of formula $H \wedge z$ above to PQE as follows. Let F be the set of all clauses of H with literal $\overline{z}$. We will refer to such clauses as **$\overline{z}$-clauses** of H. Let $G = H \setminus F$. Checking the satisfiability of $H \wedge z$ is equivalent to solving the PQE problem of finding formula $F^*(z)$ such that $F^*(z) \wedge \exists W[G] \equiv \exists W[F \wedge G]$ where $W = X \cup Y$. If $F^*(z) \equiv 1$, *i.e.*, if F^* consists of an empty set of clauses, formula $H \wedge z$ is satisfiable. If $F^*(z) = \overline{z}$, then $H \wedge z$ is unsatisfiable. In other words, if all $\overline{z}-$clauses are redundant in $\exists W[H]$, then $H \wedge z$ is satisfiable. However, if making the original $\overline{z}$-clauses of H redundant requires derivation and adding to H clause $\overline{z}$, then $H \wedge z$ is unsatisfiable.

Indeed, if clause $\overline{z}$ is derived from H it can be resolved with clause z of $H \wedge z$ to produce an empty clause. This proves the unsatisfiability of $H \wedge z$. If $\overline{z}$-clauses are redundant in $\exists W[H]$ without derivation of $\overline{z}$, the fact that H is satisfiable, implies that assignment $z = 1$ can be extended to an assignment satisfying H. This assignment obviously satisfies $H \wedge z$.

Note that in case $H \wedge z$ is unsatisfiable, the final goal of a PQE-algorithm is the same as that of a SAT-solver: the PQE-algorithm derives a clause $\overline{z}$ that is only one resolution operation away from producing an empty clause. However, in case $H \wedge z$ is satisfiable, there is an important difference: a PQE-algorithm can prove satisfiability by showing that the $\overline{z}$-clauses of H are redundant *without finding a satisfying assignment.*

2.3 Reducing Circuit-SAT to PQE: Second Method

Here we give a different method of reducing Circuit-SAT to PQE. We will refer to the methods of the previous and current subsections as first and second method respectively. The second method is to solve the PQE problem of finding a CNF formula $K(X)$ such that $K \wedge \exists V[H] \equiv \exists V[\overline{z} \wedge H]$ where $V = Y \cup \{z\}$. That is K is obtained by taking clause $\overline{z}$ out of the scope of quantifiers. It is not hard to show that $H \wedge z$ is satisfiable if and only if formula K contains at least one clause. Every complete assignment to X falsifying K specifies an input for which N evaluates to 1, *i.e.*, a counterexample. So if finding one counterexample suffices, one can stop as soon as a clause is added to K.

Notice that the first and second methods are, in a sense, *complementary*. To prove unsatisfiability of $H \wedge z$ by the first method, one needs to produce an explicit derivation of an empty clause. However, proving satisfiability of $H \wedge z$ does not require finding an explicit satisfying assignment. In the second method, the situation is the opposite. Proving satisfiability requires generating at least one clause of H and hence finding at least one counterexample. On the other hand, the fact that clause $\overline{z}$ is redundant in $\exists V[\overline{z} \wedge H]$ means that $H \wedge z$ is unsatisfiable. However, the second method does not give an explicit proof of this fact (*e.g.*, it does not generate an empty clause).

An interesting feature of the second method is that it provides a *derivation* of a counterexample. Usually a counterexample is a result of guesswork even in a formal verification tool. For example, finding a satisfying assignment by a SAT-solver requires *guessing* the decision assignments. (Implied assignments are derived from learned clauses and do not need guesswork.) This makes it hard to measure the *complexity* of finding a counterexample. In the second method, a counterexample $\boldsymbol{x}$ is a complete assignment falsifying a clause C of K. This clause is *derived* from $\overline{z} \wedge H$ and the length of this derivation can be used to measure the complexity of finding counterexample $\boldsymbol{x}$.

3 Quantifier Elimination By Dependency Sequents

In this section, we give the high-level view of our algorithms for QE and PQE.

Suppose that one needs to eliminate quantifiers from formula $\exists X[H]$. In [2], we developed a QE algorithm based on the notion of a Dependency Sequent (D-sequent). This algorithm is called *DCDS* (Derivation of Clause D-sequents). *DCDS* is based on the following two ideas. First, if one adds to H a "sufficient" number of resolvent clauses, all X-clauses (*i.e.*, clauses containing variables of X) will become redundant. Second, proving clause redundancy globally is hard. So it makes sense to use branching to prove redundancy of X-clauses in subspaces first and then merge the results of different branches. Proving redundancy of X-clauses of H in subspaces, in general, requires adding resolvent clauses to H.

Let $\boldsymbol{q}$ be an assignment to variables of H. A record $(\exists X[H], \boldsymbol{q}) \rightarrow R$ called **D-sequent** is used by *DCDS* to store the fact that a set R of X-clauses is redundant in $\exists X[H]$ in subspace $\boldsymbol{q}$. Assignment $\boldsymbol{q}$ is called the conditional part

of the D-sequent. When *DCDS* merges results of branching on a variable v, it "merges" D-sequents obtained in subspaces $v = 0$ and $v = 1$ using a resolution-like operation called *join*. This results in producing new D-sequents that do not have an assignment to variable v in their conditional parts. The objective of *DCDS* is to derive D-sequent $(\exists X[H], \emptyset) \rightarrow H^X$ where H^X is the set of all X-clauses of H. This D-sequents states unconditional redundancy of X-clauses in $\exists X[H]$. Once this D-sequent is derived, a solution to the QE problem is obtained by removing all X-clauses from H.

We have developed a PQE algorithm based on *DCDS*. This algorithm is called ***DS-PQE*** (DS stands for D-Sequents). Suppose one needs to solve the PQE problem of taking F out of the scope of quantifiers in $\exists X[F \wedge G]$. *DS-PQE* is based on the same two ideas as above. The main difference of *DS-PQE* from *DCDS*, is that the former needs to prove only the redundancy of X-clauses of F. So the objective of *DS-PQE* is to derive D-sequent $(\exists X[F \wedge G], \emptyset) \rightarrow F^X$ where F^X is the set of all X-clauses of F. In *DS-PQE*, new resolvent clauses are assumed to be added to F while G stays unchanged. So after the final D-sequent above is derived, a solution to the PQE problem is obtained from F by discarding the clauses of F^X.

4 Software Description

DS-PQE is implemented as a stand-alone program written in C++. *DS-PQE* accepts formula $\exists X[F \wedge G]$ and returns formula F^* such that $F^* \wedge \exists X[G] \equiv \exists X[F \wedge G]$. The formula $\exists X[F \wedge G]$ is specified by three files. The first file describes the CNF formula $F \wedge G$ in the DIMACS format. The second file contains the free variables of $\exists X[F \wedge G]$. Variables not mentioned in this file are assumed to be quantified. The third file contains the clauses of F. The resulting CNF formula F^* is returned in a file in the DIMACS format.

5 Acknowledgments

This research was supported in part by DARPA under AFRL Cooperative Agreement No. FA8750-10-2-0233 and by NSF grants CCF-1117184 and CCF-1319580.

References

1. A. R. Bradley. Sat-based model checking without unrolling. In *VMCAI*, pages 70–87, 2011.
2. E.Goldberg and P.Manolios. Quantifier elimination via clause redudnancy. In *FMCAD-13*, pages 85–92, 2013.
3. E.Goldberg and P.Manolios. Partial quantifier elimination. Submitted for publication, 2014.
4. K. L. Mcmillan. Interpolation and sat-based model checking. In *CAV-03*, pages 1–13. Springer, 2003.