GhostQ QBF Solver System Description

William Klieber

Carnegie Mellon University
Pittsburgh, Pennsylvania, USA

April 25, 2012

1 Overview

GhostQ is a DPLL-based solver that uses *ghost variables* to achieve symmetry in handling of universal and existential variables. The original version of GhostQ [4] has support for non-prenex instances, but the version described here drops support for non-prenex instances and instead optimizes for prenex instances.

The present version of GhostQ has three modes of operation. In the first mode, GhostQ behaves largely as described in our original paper [4], but with several additional low-level optimizations. In the second mode, CEGAR learning is enabled, as described in [3]. The third mode is like the second mode, except that we first run bloqqer [1] on the input file.[1]

2 Ghost Literals

Goultiaeva et al. [2] introduce a powerful propagation technique for QBF that significantly improves on existing QBF solvers on a variety of benchmarks. With their technique, if the solver notices that a gate literal g must be true in order for the existential player to win, then g becomes forced. However, this technique is asymmetric between the existential and universal players. A gate literal g is forced if it is needed for the existential player to win, but not if it is needed for the universal player to win. We adapt this technique so that the universal variables benefit from the same propagation technique as do the existential variables and so that the learning procedure for satisfying assignments is just as powerful as for falsifying assignments.

In our prenex solver, for each gate variable g, we introduce two *ghost* variables, $g\langle\mathtt{U}\rangle$ for Player `U` and $g\langle\mathtt{E}\rangle$ for Player `E`. A ghost literal $g\langle P\rangle$ is be forced whenever we detect that Player P cannot win unless g is made true.

3 CEGAR Learning

We modify GhostQ by inserting a call to a CEGAR-learning procedure after performing standard DPLL learning. We write "Φ_{in}" to denote the current

[1] If bloqqer sufficiently simplifies the problem, we use the output of bloqqer. Otherwise we discard it and proceed with the original input file, since the reverse engineering code employed by GhostQ currently cannot handle the output of bloqqer.

1. Let X_c be the quantifier block of the last decision literal.
   Let Q_c and Φ_c be such that $(Q_c X_c.\Phi_c)$ is a subformula of Φ_{in}.
2. Let π_c be a complete assignment for X_c created by extending the solver's current assignment with arbitrary values for the unassigned variables in X_c and removing variables in blocks other than X_c. This assignment π_c corresponds to the *counterexample* in the recursive CEGAR approach.
3. We modify Φ_{in} by:
   - substituting $(\exists X_c.\Phi_c)$ with $(\exists X_c.\Phi_c) \vee \Phi_c[\pi_c]$, if $Q_c =$ "$\exists$", or
   - substituting $(\forall X_c.\Phi_c)$ with $(\forall X_c.\Phi_c) \wedge \Phi_c[\pi_c]$, if $Q_c =$ "$\forall$".
4. All variables that are bound by a quantifier inside $\Phi_c[\pi_c]$ are renamed to preserve uniqueness of variable names.

Fig. 1. CEGAR Learning in DPLL

input formula, i.e., the input formula enhanced with what the solver has learned up to now. Both standard DPLL learning and CEGAR learning are performed by modifying Φ_{in}. CEGAR learning is performed only if the last decision literal is owned by the winner. (The case where the last decision literal is owned by the losing player corresponds to the conflicts that take place *within* the underlying SAT solver in RAReQS.) The CEGAR-learning procedure is shown in Figure 1.

3.1 CEGAR Implementation Details

We have implemented a limited version of CEGAR learning in the solver GhostQ. Our implementation uses a modified version of step 3 of Figure 1. We substitute π_c into the original version of the input formula Φ_{in}, not the current version of Φ_{in}. Although substituting into the original formula instead of the current formula potentially reduces the effectiveness of CEGAR learning (since we can't learn a refinement of a refinement), it reduces the memory consumed per refinement. Unit propagation and the Pure Literal Rule are applied to simplify the result of the substitution, among other optimizations.

References

1. A. Biere, F. Lonsing, and M. Seidl. Blocked Clause Elimination for QBF. In *CADE*, 2011.
2. A. Goultiaeva, V. Iverson, and F. Bacchus. Beyond CNF: A Circuit-Based QBF Solver. In *SAT 2009*.
3. M. Janota, W. Klieber, J. Marques-Silva, and E. Clarke. Solving QBF with Counterexample Guided Refinement. In *SAT 2012, to appear*.
4. W. Klieber, S. Sapra, S. Gao, and E. M. Clarke. A Non-prenex, Non-clausal QBF Solver with Game-State Learning. In *SAT 2010*.