Binary Clause Reasoning in QBF

Horst Samulowitz[1] and Fahiem Bacchus[1]

Department of Computer Science, University Of Toronto,*
Toronto, Ontario, Canada
[horst|fbacchus]@cs.toronto.edu

Abstract. Binary clause reasoning has found some successful applications in SAT, and it is natural to investigate its use in various extensions of SAT. In this paper we investigate the use of binary clause reasoning in the context of solving Quantified Boolean Formulas (QBF). We develop a DPLL based QBF solver that employs extended binary clause reasoning (hyper-binary resolution) to infer new binary clauses both before and during search. These binary clauses are used to discover additional forced literals, as well as to perform equality reduction. Both of these transformations simplify the theory by removing one of its variables. When applied during DPLL search this stronger inference can offer significant decreases in the size of the search tree, but it can also be costly to apply. We are able to show empirically that despite the extra costs, binary clause reasoning can improve our ability to solve QBF.

1 Introduction

DPLL based SAT solvers standardly employ only unit propagation during search. Unit propagation has the advantage that it can be very efficiently implemented, but at the same time it is relatively limited in its inferential power. The more powerful inferential mechanism of reasoning with binary clauses has been investigated in [1, 2]. In particular, Bacchus [2] demonstrated that by using a rule of hyper-binary resolution, which allows the binary clause subtheory to be clashed against its non-binary counterpart, binary clause reasoning can be very effective in pruning the size of the search space. It can also be dramatically effective in decreasing the time required to solve SAT problems, but not always.

The difficulty arises from the extra time required to perform binary clause reasoning, which tends to scale non-linearly with the size of the SAT theory. Hence, on very large SAT formulas, binary clause reasoning is often not cost effective. QBF instances, on the other hand, are generally much smaller than SAT instances. First, QBF allows a much more compact representation of many problems, so problems that would be very large in SAT can be quite small when represented in QBF. Second, QBF is in practice a much harder problem than SAT, so it is unlikely that "solvable" instances will ever be as large as solvable SAT instances. This makes the application of extensive binary clause reasoning more attractive on QBF instances, since such reasoning is more efficient on smaller theories.

In this paper we investigate using binary clause reasoning with QBF. We find that our intuition that such reasoning might be useful for QBF to be empirically true. How-

* This research was supported by the Canadian Government through their NSERC program.

ever, we also find that there are a number of issues arising from the use of such reasoning. First, there are some issues involved in employing such reasoning soundly in a QBF setting. We describe these issues and show how they can be resolved. Second we have found that such reasoning does not universally yield an improvement. Instead one has to be careful about when and where one employs such reasoning.

We have found that binary clause reasoning to be almost universally useful prior to search when used in a QBF preprocessor (akin to the SAT preprocessor of [3]), and we present a more detailed description of preprocessing in [4]. To study the dynamic use of binary clause reasoning during search we have implemented a QBF solver that performs binary clause reasoning at every node of the search tree. Our empirical results indicate that binary clause reasoning can be effective when used dynamically. However, it is not as uniformly effective as it is in a preprocessor context. We provide some insights as to when it can be most useful applied dynamically.

In the rest of the paper we first provide some background, then we discuss how binary clause reasoning can be soundly employed in QBF. We then demonstrate that binary clause reasoning is effective in improving our ability to solve QBF instances. Part of that improvement actually occurs prior to search, and we briefly discuss our findings on this point. These empirical observations lead to the development of a preprocessor for QBF that we describe in [4]. Then we investigate the dynamic use of binary clause reasoning, and show that it also can be effective in the dynamically, but not universally so. Our overall conclusion is that binary clause reasoning does have an important role to play in solving QBF but that further investigation is required to isolate more precisely where it can be most effectively applied.

2 Background

2.1 QBF

A quantified boolean formula (QBF) has the form $\boldsymbol{Q}.F$, where F is a propositional formula expressed in CNF and $\boldsymbol{Q}$ is a sequence of quantified variables ($\forall x$ or $\exists x$). We require that no variable appear twice in $\boldsymbol{Q}$, that F contains no free variables, and that $\boldsymbol{Q}$ contains no extra or redundant variables.

A **quantifier block** qb of $\boldsymbol{Q}$ is a maximal contiguous subsequence of $\boldsymbol{Q}$ where every variable in qb has the same quantifier type. We order the quantifier blocks by their sequence of appearance in $\boldsymbol{Q}$: $qb_1 \leq qb_2$ iff qb_1 is equal to or appears before qb_2 in $\boldsymbol{Q}$. Each variable x in F appears in some quantifier block $qb(x)$, and the ordering of the quantifier blocks imposes a partial order on the variables. For two variables x and y we say that $x \leq_q y$ iff $qb(x) \leq qb(y)$. The variables in the same quantifier block are unordered. We also say that x is **universal** (**existential**) if its quantifier in $\boldsymbol{Q}$ is $\forall$ ($\exists$).

For example, $\exists e_1 e_2.\forall u_1 u_2.\exists e_3 e_4.(e_1, \neg e_2, u_2, e_4) \wedge (\neg u_1, \neg e_3)$ is a QBF with $\boldsymbol{Q} = \exists e_1 e_2.\forall u_1 u_2.\exists e_3 e_4$ and $F = (e_1, \neg e_2, u_2, e_4) \wedge (\neg u_1, \neg e_3)$. The quantifier blocks in order are $\exists e_1 e_2$, $\forall u_1 u_2$, and $\exists e_3 e_4$, the u_i variables are universal while the e_i variables are existential, and $e_1 \leq_q e_2 <_q u_1 \leq_q u_2 <_q e_3 \leq_q e_4$.

The **restriction** of a formula $\boldsymbol{Q}.F$ by a literal ℓ (denoted by $\boldsymbol{Q}.F|_\ell$) is the new formula $\boldsymbol{Q}'.F'$ where F' is F with all clauses containing ℓ removed and $\bar{\ell}$, the negation of ℓ, removed from all remaining clauses, and $\boldsymbol{Q}'$ is $\boldsymbol{Q}$ with the variable of ℓ and its quantifier removed. For example, $\big(\forall xz.\exists y.(\bar{y}, x, z) \wedge (\bar{x}, y)\big)|_{\bar{x}} = \forall z.\exists y(\bar{y}, z)$.

Semantics. A SAT model $\mathcal{M}_s$ of a CNF formula F is a truth assignment π to the variables of F that satisfies every clause in F. In contrast a QBF model (**Q-model**) $\mathcal{M}_q$ of a quantified formula $\boldsymbol{Q}.F$ is a **tree** of truth assignments in which the root is the empty truth assignment, and every node n assigns a truth value to a variable of F not yet assigned by one of n's ancestors. The tree $\mathcal{M}_q$ is subject to the following conditions:

1. For every node n in $\mathcal{M}_q$, n has a sibling if and only if it assigns a truth value to a universal variable x. In this case it has exactly one sibling that assigns the opposite truth value to x. Nodes assigning existentials have no siblings.
2. Every path π in $\mathcal{M}_q$ (π is the sequence of truth assignments made from the root to a leaf of $\mathcal{M}_q$) must assign the variables in an order that respects $<_q$. That is, if n assigns x and one of n's ancestors assigns y then we must have that $y \leq_q x$.
3. Every path π in $\mathcal{M}_q$ must be a SAT model of F.

Thus a Q-model has a path for every possible setting of the universal variables of $\boldsymbol{Q}$, and each of these paths is a $\leq_q$ ordered SAT model of F. We say that $\boldsymbol{Q}.F$ is QSAT iff it has a Q-model. The QBF problem is to determine whether or not $\boldsymbol{Q}.F$ is QSAT.

A more standard way of defining QSAT is the recursive definition: (1) $\forall x\boldsymbol{Q}.F$ is QSAT iff both $\boldsymbol{Q}.F|_x$ and $\boldsymbol{Q}.F|_{\bar{x}}$ are, and (2) $\exists x\boldsymbol{Q}.F$ is QSAT iff at least one of $\boldsymbol{Q}.F|_x$ and $\boldsymbol{Q}.F|_{\bar{x}}$ is. By removing the quantified variables one by one, in $\leq_q$ order, we arrive at either a QBF with an empty clause in its body F (which is not QSAT) or a QBF with an empty body F (which is QSAT). These two definitions are provably equivalent.

The advantage of our "tree-of-models" definition is that it makes the following observations more apparent.

A. If F' has the same satisfying assignments (SAT models) as F then $\boldsymbol{Q}.F$ will have the same satisfying models (Q-models) as $\boldsymbol{Q}.F'$. **Proof**: $\mathcal{M}_q$ is a Q-model of $\boldsymbol{Q}.F$ iff each path in $\mathcal{M}_q$ is a SAT model of F iff each path is a SAT model of F' iff $\mathcal{M}_q$ is a Q-model of $\boldsymbol{Q}.F'$. This observation allows us to transform F with any model preserving SAT transformation. Note that the transformation must be model preserving, i.e., it must preserve all SAT models of F. Simply preserving whether or not F is SAT is not sufficient.

B. A Q-model preserving (but not SAT model preserving) transformation that can be performed on $\boldsymbol{Q}.F$ is **universal reduction** (UR) [5]. A universal variable u is called a *tailing universal* in a clause c if for every existential variable $e \in c$ we have that $e <_q u$. The universal reduction of a clause c is the process of removing all tailing universals from c. UR preserves the set of Q-models. This can be seen by observing that any path in a Q-model must satisfy the universally reduction of every clause in the theory: if it doesn't then another path of the Q-model will falsify c contradicting the fact that it is a Q-model.

2.2 Hyper-Binary Resolution and Equality Reduction

Now we recall the techniques for binary clause reasoning in SAT first presented in [2, 3]. We first define the hyper-binary resolution (*HypBinRes*) rule of inference that generates new binary unit clauses.

Definition 1 (*HypBinRes*). *Given a single n-ary clause $c = (l_1, l_2, ..., l_n)$, D a subset of c, and the set of binary clauses $\{(\ell, \bar{l})|l \in D\}$, infer the new clause $b = (c - D) \cup \{\ell\}$ **if** b is either **binary** or **unary**.*

For example, from (a, b, c, d), $(h, \bar{a})$, $(h, \bar{c})$ and $(h, \bar{d})$, we infer the new binary clause (h, b), similarly from (a, b, c) and $(b, \bar{a})$ the rule generates (b, c). The rule also covers the standard case of resolving two binary clauses (from (l_1, l_2) and $(\bar{l}_1, \ell)$ infer (ℓ, l_2)) and it can generate unit clauses (e.g., from $\{(l_1, \ell), (\bar{l}_1, \ell)\}$ we infer $(\ell, \ell) \equiv (\ell)$). *HypBinRes* is a hyper-resolution step because it collapses in one step a sequence of ordinary resolution steps.

The advantage of *HypBinRes* inference is that it does not blow up the theory (it can only add binary or unary clauses to the theory) and it can discover a lot of new unit clauses. These unit clauses can then be used to simplify the formula by doing unit propagation which in turn might allow more applications of *HypBinRes*. Applying *HypBinRes* and unit propagation until closure (i.e., until nothing new can be inferred) uncovers *all* failed literals. That is, in the resulting reduced theory there will be no literal ℓ such that forcing ℓ to be true followed by unit propagation results in a contradiction. This and other results about *HypBinRes* are proved in the above references.

In addition to uncovering unit clauses we can use the binary clauses to perform equality reductions. In particular, if we have two clauses $(\bar{x}, y)$ and $(x, \bar{y})$ we can replace all instances of y in the formula by x (and $\bar{y}$ by $\bar{x}$) or all instances of x by y. This might result in some tautological clauses which can be removed, and some clauses which are reduced in length because of duplicate literals. Such reductions might enable further *HypBinRes* inferences.

Taken together *HypBinRes* and equality reduction (*HypBinRes+eq*) can significantly reduce a SAT formula removing many of its variables and clauses. Such inference can be applied prior to search in a preprocessor, and as shown in [3] this can yield significant reductions in the number of variables and clauses in a theory. One can also incrementally maintain *HypBinRes+eq* closure during search as it is done in [2].

To maintain *HypBinRes+eq* closure during search we must trigger the *HypBinRes* inference step incrementally. It would be too expensive to continually search exhaustively for possible new applications of *HypBinRes*. During search the formula is restricted by literals that we choose to make true, or that are forced by unit propagation. This gives rise to only two different opportunities for additional applications of *HypBinRes*. First, if a k-ary clause is reduced to a binary clause the new binary clause might enable new *HypBinRes* steps. Second, when k-ary clauses are reduced in size it is possible that a previously existing set of binary clauses can generate an *HypBinRes* inference that was not available on the longer clause. For example, if we have the n-ary clauses $(h, \bar{d}, x)$ (a, b, c, d) and the binary clauses $(h, \bar{a})$, $(h, \bar{c})$ no *HypBinRes* inference is possible. A new *HypBinRes* inference could be applied if either we make x false generating a new binary clause $(h, \bar{d})$, or if we make d false reducing the clause (a, b, c, d) to (a, b, c) against which the existing binary clauses can be resolved. The dynamic *HypBinRes* solver described in [2] kept track of these two types of situations, testing only these situations for new possible *HypBinRes* steps.

2.3 Hyper-Binary Resolution and Equality Reduction in QBF

There are two problems with employing *HypBinRes+eq* in the context of QBF. First, it is not sound for QBF unless some additional restrictions are applied. Second, it misses out on some important additional inferences that can be achieved through universal reduction. We elaborate on these two issues.

Given a QBF $\boldsymbol{Q}.F$ applying *HypBinRes+eq* and unit propagation to F results in a formula F'. However, the new QBF formula $\boldsymbol{Q}'.F'$ might not be Q-equivalent to $\boldsymbol{Q}.F$ (where $\boldsymbol{Q}'$ is $\boldsymbol{Q}$ with all variables not in F' removed), so this straightforward approach to using *HypBinRes* is not sound. The problem here is that F' does not have exactly the same SAT models as F so condition **A** above does not apply. In particular, the models of F' do not make assignments to variables that have been removed by unit propagation and equivalence reduction. Hence, a Q-model of $\boldsymbol{Q}'.F'$ might not be extendable to a Q-model of $\boldsymbol{Q}.F$. For example, if a universal variable in F was forced, then $\boldsymbol{Q}'.F'$ might be QSAT, but $\boldsymbol{Q}.F$ is not—no Q-model of $\boldsymbol{Q}.F$ can exist since no path that sets the forced universal to its opposite value can be a SAT model of F.

Making unit propagation sound for QBF is quite simple. In particular, unit propagation only causes a problem when a universal variable is forced. We can deal with this by regarding the unit propagation of a universal variable as the derivation of a failure (i.e., the derivation of an empty clause).

Making equality reduction sound for QBF is a bit more subtle. Consider a formula F in which we have the two clauses $(x, \bar{y})$ and $(\bar{x}, y)$. Since every path in any Q-model satisfies F, this means that along any path x and y must have the same truth value. However, in order to soundly replace all instances of one of these variables by the other in F, we must respect the quantifier ordering. In particular, if $x <_q y$ then we must replace y by x. We call this *$<_q$-preferred equality reduction*. It would be unsound to do the replacement in the other direction. For example, say that x appears in quantifier block 3 while y appears in quantifier block 5 with both x and y being existential. The binary clauses above will enforce the constraint that along any path of any Q-model once x is assigned y must get the same value. In particular, y will be invariant as we change the assignments to the universal variables in quantifier block 4. This constraint will continue to hold if we replace y by x in all of the clauses of F. However, if we perform the opposite replacement, we would be able to make y vary as we vary the assignments to the universal variables in block 4: i.e., the opposite replacement would weaken the theory perhaps changing the formula's Q-SAT status. The same reasoning holds if x is universal and y is existential. However, if y is universal the two binary clauses imply that we will never have the freedom to assign y both of its values irrespective of the assignment of x. That is, in this case the QBF is UNQSAT, and we can again treat this case as if the empty clause has been derived.

These considerations suffice to make *HypBinRes+eq* sound for QBF. However, they remain weaker than they should be. To achieve more powerful inference we must take into account universal reduction. In particular, we can apply the following modification of *HypBinRes* that "folds" UR into the inference rule.

Definition 2 (*HypBinRes+UR*). *Given a single n-ary clause $c = (l_1, l_2, ..., l_n)$, D a subset of c, and the set of binary clauses $\{(\ell, \bar{l}) | l \in D\}$, infer the universal reduction of the clause $(c - D) \cup \{\ell\}$ if this reduction is either binary or unary.*

For example, from $(u_1, e_3, u_4, e_5, u_6, e_7)$, $(e_2, \bar{e}_7)$, $(e_2, \bar{e}_5)$ and $(e_2, \bar{e}_3)$ we infer the new binary clause (u_1, e_2) when $u_1 \leq_q e_2 \leq_q e_3 \leq_q u_4 \leq_q e_5 \leq_q u_6 \leq_q e_7$. This example also shows that *HypBinRes+UR* is able to derive clauses that *HypBinRes* cannot. Since clearly *HypBinRes+UR* can derive anything *HypBinRes* can, *HypBinRes+UR* is a more powerful rule of inference. It should be noted that UR cannot be applied after

2clsQ($\boldsymbol{Q}.F$, $Level$)
  if F contains an [empty clause/is empty]
    Compute a new [clause/cube] and backtrack level btL by [conflict/solution] analysis
    return([FALSE/TRUE], btL)
  Pick a variable v from the outermost quantifier block
  for $\ell \in \{v, \bar{v}\}$
    $\boldsymbol{Q}'.F' = \boldsymbol{Q}.F|_\ell$ reduced by *HypBinRes+UR*, equality reduction, unit propagation, and universal reduction
    $(Succ, btL) =$ 2clsQ($\boldsymbol{Q}'.F'$, $Level + 1$)
    if $btL <$ Level ***return***($Succ, btL$)
  if v is [universal/existential]
    Compute new [cube/clause] from the [cubes/clauses] learned from v and $\bar{v}$ by resolution
    Compute backtrack level btL from new [cube/clause]
    return([TRUE/FALSE], btL)

Fig. 1. 2clsQ Algorithm. Invoked with original QBF and *Level=1*. Returns (TRUE, 0) indicating QSAT or (FALSE, 0) indicating UNQSAT.

HypBinRes as *HypBinRes* can only generate binary clauses. Instead UR must be folded into the *HypBinRes* rule as we have specified here.

Interestingly, once we add UR to *HypBinRes* many of the issues we had with soundness automatically resolve themselves, and we obtain the following result:

PROPOSITION 1 *Let F' be the result of applying HypBinRes+UR, unit propagation, UR (i.e., UR outside of HypBinRes as well as inside), and $<_q$ preferred equality reduction to F until closure. Then the Q-models of $\boldsymbol{Q}'.F'$ are in 1-1 correspondence with the Q-models of $\boldsymbol{Q}.F$.*

The only further constraint is that UR must be applied prior to unit propagation. In particular, if we have a unit clause containing a single universal variable, we should not unit propagate that universal. Rather we should immediately apply UR to obtain the empty clause.

As an example of how applying UR resolves some of the soundness issues mentioned above, consider the case where we have the two binary clauses $(x, \bar{y})$ and $(\bar{x}, y)$ with $x <_q y$. As pointed out above, when y is universal we have an immediate failure. In fact, applying universal reduction detects this failure: after UR we obtain the two clauses (x) and $(\bar{x})$ which immediately resolve to the empty clause. Hence, this proposition tells us that we can apply *HypBinRes+UR+eq* in QBF quite cleanly: we simply have to restrict equality reduction to respect the quantifier ordering and give precedence to UR over unit propagation.

3 2clsQ

We have implemented *HypBinRes+UR+eq* in a DPLL based QBF solver by modifying the 2clsEq SAT solver [2]. The resulting QBF solver, 2clsQ, performs *HypBinRes+UR+eq* reasoning at every node of the search tree. An abstract outline of its algorithm is shown in Figure 1. The following changes were made to the 2clsEq SAT solver to make it into a QBF solver. First, branching had to be constrained so that the quantifier ordering is respected. Second, equality reduction had to be modified so that it respects the quantifier ordering. In the 2clsEq implementation an entire set of variables could

be detected to be equivalent at once, so we must pick a variable v from the outermost quantifier block among that set and then replace all of the other variables with v.

Third, we had to modify the code that tested for possible new applications of *HypBinRes* to account for universal reduction. When a new binary clause (x, y) is generated we can continue to test all clauses containing $\bar{x}$ as well as all clauses containing $\bar{y}$ to see if this new binary clause triggers any new applications of *HypBinRes+UR*. For example, if $\bar{x} \in c$, we determine the set S of other literals $\ell \in c$ that can be resolved away from c by binary clauses of the form $(y, \bar{\ell})$. Then we check if $c - S$ can be universally reduced to a clause of length 2 or less. The other trigger for new applications of *HypBinRes* occurs when a k-ary clause has been reduced in size, as discussed above. Unfortunately, this situation is relatively expensive to extend to *HypBinRes+UR*. With just *HypBinRes* when a clause c has just been reduced in size to length i, we only need to look for a literal x such that there are $i-1$ binary clauses $(x, \bar{\ell})$ with $\ell \in c$. From these clauses we can then infer a new binary clause (x, y), where $y \in c$ is the single literal not covered in the set of clauses $(x, \bar{\ell})$. This can be accomplished relatively efficiently by first taking any two literals of c, l_1 and l_2 and examining the set of literals $L = \{y|$ either $(y, \bar{l}_1)$ or $(y, \bar{l}_2)$ exists$\}$. We then know that any literal x satisfying the above condition must be in L—any such literal must have a binary clause with one of $\bar{l}_1$ or $\bar{l}_2$—and we can restrict our attention to the literals in L.

Unfortunately, this strategy for limiting the set of literals to examine for potential new *HypBinRes* steps against a clause breaks down when we move to *HypBinRes+UR*. For example, consider the clauses $c = (e_1, u_1, u_2, u_3, e_2, u_4, u_5, e_3)$, $(e, \bar{e}_2)$, $(e, \bar{e}_3)$ with $e <_q e_1 <_q u_1, u_2, u_3 <_q e_2 <_q u_4, u_5 <_q e_3$. We can infer the new binary clause (e_1, e) by applying *HypBinRes+UR*. In this case, the literal e has only two binary clauses that can resolve against c, and so it does not fall into the set L defined above. Hence, it is not possible to limit our attention to the literals in L. It is still possible to detect all possible *HypBinRes+UR* inferences available from c in polynomial time, but it becomes more expensive to do so. Hence, in our implementation we do only a partial, and cheaper, test for new *HypBinRes+UR* inference on k-ary clauses that have been reduced in size. That is, we do not achieve *HypBinRes+UR* closure in 2clsQ.

Fourth, the algorithm employs both conflict and solution analysis for learning new clauses and solution cubes. Since literals can be forced from an extensive combination of binary clause reasoning and equality reduction, it was very difficult to implement 1-UIP clause learning. Instead, 2clsQ learns 'all decision clauses' [6]. The learned clauses are used to enhance unit propagation. However, we do not perform *HypBinRes+UR* or equality reduction against them as this appears to be too expensive. Solution analysis (cube learning) is done in the manner introduced in [6, 7]. The learned cubes are also used to prune branches in the search. In particular, when a universal variable is set this might trigger a cube making search below that setting unnecessary.

Finally, we modified the original 2clsEq branching heuristics to take into account the varying nature of QBF search. In our implementation we combined two branching heuristics in the following way. Whenever 2clsQ encounters a conflict we try to generate more conflicts by branching on variables that cause the largest number of unit propagations (under *HypBinRes* this number is equal to the number of binary clauses the variable appears in). On the other hand when 2clsQ finds a solution we try to gen-

erate more solutions by branching on variables that will satisfy the most clauses. Thus the branching heuristic switches dependent on what "mode" the search is in.

4 Empirical Results

To evaluate the empirical effect of binary clause reasoning we considered all of the non-random benchmark instances from QBFLib (2005) [8] (508 instances in total). We discarded the instances from the benchmark families von Neumann and Z since these can all be solved very quickly by any state of the art QBF solver (less than 10 sec. for the entire suite of instances). We also discarded the instances in benchmark families Uclid, Jmc, and Jmc-squaring. None of these instances can be solved within a time bound of 5,000 seconds by any of the QBF solvers we tested. This left us with 465 instances from 18 different benchmark families. We tested all of these instances.

We tested 2clsQ [9] along with five other state of the art QBF solvers **Quaffle** [7] (version as of Feb. 2005), **Quantor** [10] (version as of 2004), **Qube** (release 1.3) [11], **Skizzo** [12] (release 0.82) and **SQBF** [13]. Quaffle, Qube and SQBF are based on search, whereas Quantor is based on variable elimination. Skizzo uses mainly a combination of variable elimination and search, but it also applies a variety of other kinds of reasoning on the symbolic and the ground representations of the instances.[1] All tests were run on a Pentium 4 3.60GHz CPU with 6GB of memory. The time limit for each run of any of the solvers was set to 5,000 seconds.

Table 1 shows the performance of 2clsQ and the other five solvers on the 465 problem instances we tested. The table is broken down by benchmark family as the structural properties of the families can be quite distinct. This structural distinctions are reflected in fact that the "best" solver for each family varies widely, where we measure best by the success rate of the solver on that families' instances breaking ties by CPU time consumed. By this measurement 2clsQ is best on 3 families, which is better than any other search based solver (Quaffle, Qube, and SQBF), but not as good as Skizzo which is best on 8 families. Another comparison is to examine the average success rate over all benchmark families, shown in the final row of the table. A high average displays fairly robust performance across structurally distinct instances. On this measure 2clsQ is again superior to the other search based solvers with an average success rate of 58%, higher than any of the other search based solvers, but again not as good as Skizzo or Quantor. In terms of CPU time, the search based solvers are roughly comparable over their solvable instances, but both Quantor and Skizzo are notably faster.

Our first results lead to the following conclusions. Binary clause reasoning improves search based solvers, but the non-search solver Quantor and the mixture of search and variable elimination employed in Skizzo often have superior performance. The superior performance of Skizzo indicates that mixing search and variable elimination (as done by Skizzo) is very effective. We also observe that both Quantor and Skizzo are still inferior to *some* search based solver on 43% of the families. Furthermore, if we examine those cases where a solver is able to achieve a strictly higher success rate than any other solver (indicating that it can solve some instances not solvable by any of the other solvers),

[1] Skizzo also employs some binary clause reasoning and equality reduction. But hyper binary resolution is not used, non-binary clauses are not involved in the inference steps ([12] incorrectly claims that hyper-binary resolution is used).

Benchmark Families	*2clsQ*		*Quaffle*		*Qube*		*SQBF*		*Quantor*		*Skizzo*	
(# instances)	*Succ. %*	*time*	*Succ. %*	*time*	*Succ. %*	*time*	*Succ. %*	*time*	*Succ. %*	*time*	*Succ. %*	*time*
ADDER (16)	44%	5,267	13%	1	19%	72	13%	3	25%	25	**50%**	955
adder (16)	19%	0	44%	5	**44%**	0	38%	2,677	25%	30	44%	454
Blocks (16)	50%	46	75%	1,284	69%	1,774	75%	2,043	**100%**	308	69%	2,068
C (24)	21%	16	21%	5,356	8%	4	17%	4,741	21%	140	**25%**	1,070
Chain (12)	**100%**	0	67%	6,075	83%	4,990	58%	4,192	**100%**	0	100%	1
Connect (60)	**100%**	7	70%	254	75%	7,013	67%	0	67%	14	68%	802
Counter (24)	33%	4,319	38%	5	33%	2	38%	9	50%	217	**54%**	1,035
EV-Pursuer(38)	26%	2,836	26%	1,963	18%	4,401	**32%**	4,759	3%	74	29%	1,450
FlipFlop (10)	100%	4	**100%**	0	100%	1	80%	5,027	100%	3,260	100%	6
K (107)	35%	20,575	35%	18,451	37%	25,397	33%	5,563	64%	3,855	**88%**	2,081
Lut (5)	100%	19	**100%**	1	100%	3	100%	1,246	100%	3	100%	9
Mutex (7)	43%	22	29%	43	43%	64	43%	1	43%	0	**100%**	1
Qshifter (6)	33%	59	17%	0	33%	29	33%	1,108	100%	26	**100%**	8
S (52)	8%	9	2%	0	4%	401	2%	1	25%	910	**27%**	643
Szymanski (12)	**67%**	2,741	0%	0	8%	0	8%	1,203	25%	7	41%	1,147
TOILET (8)	75%	528	75%	61	63%	496	100%	1,308	100%	4,135	**100%**	1
toilet (38)	84%	47	97%	115	**100%**	58	97%	395	100%	684	100%	84
Tree (14)	100%	296	100%	37	**100%**	0	93%	1,051	**100%**	0	**100%**	0
Summary	58%	36,793	50%	33,653	52%	44,708	51%	35,326	64%	10,432	71%	11,817

Table 1: Percentage of each Benchmark family solved and time taken for solved instances in CPU seconds (5,000 sec. consumed by each unsolved instances is not counted). For each family the solver with highest success rate is show in bold, where ties are broken by time required to solve these instances. The summary line shows the average success rate over all benchmark families and the total time taken (on solved instances only).

we see that 2clsQ achieves this on 2 families, Quaffle on zero, Qube on zero, SQBF on one, Quantor on one, and Skizzo on 6 families. Thus we conclude that binary clause reasoning as embodied in 2clsQ has some potential in increasing our ability to solve QBF (as to the techniques embedded in SQBF, Quantor, and Skizzo).

4.1 Dynamic Binary Clause Reasoning

In SAT it was observed that binary clause reasoning could be very beneficial even when done prior to search, in a preprocessing phase [3]. Hence, a natural question was to investigate the difference between dynamic and static (i.e., before search) application of binary clause reasoning. As part of that investigation we constructed a QBF preprocessor that applies *HypBinRes+UR+eq* to simplify a QBF instance. We found that this yielded a very consistent speedup for all of the other QBF solvers, and we describe those results in more detail in [4].

Without getting into the details of our preprocessor results, we can still use our preprocessor to throw light on the effect of dynamic binary clause reasoning. In particular, we are interested in the question of how much of 2clsQ's benefits accrue from the dynamic application of binary clause reasoning. Is utilizing binary clause reasoning solely in a preprocessor sufficient, or is it also useful to use such reasoning dynamically during

Benchmark Families	*2clsQ*		*Quaffle*		*Qube*		*SQBF*		*Quantor*		*Skizzo*	
(# instances)	*Succ. %*	*time*	*Succ. %*	*time*	*Succ. %*	*time*	*Succ. %*	*time*	*Succ. %*	*time*	*Succ. %*	*time*
ADDER (16)	44%	5,267	13%	1	19%	26	13%	1	25%	26	**50%**	792
adder (16)	19%	0	44%	4	**44%**	1	38%	1,546	25%	27	44%	550
Blocks (16)	50%	46	88%	1,025	69%	242	82%	3,434	**100%**	79	88%	11
C (24)	21%	16	25%	4,947	21%	683	25%	20	29%	5,189	**29%**	1,483
Chain (12)	100%	0	100%	0	100%	0	100%	0	100%	0	100%	0
Connect (60)	100%	7	100%	7	100%	7	100%	7	100%	7	100%	7
Counter (24)	33%	4,319	38%	5	33%	1	38%	20	50%	141	**54%**	731
EV-Pursuer(38)	26%	2,836	26%	1,961	18%	2,537	32%	4,508	5%	4,809	**39%**	5,753
FlipFlop (10)	100%	4	100%	4	100%	4	100%	4	100%	4	100%	4
K (107)	35%	20,575	36%	21,446	42%	30,606	35%	12,859	83%	6,898	**91%**	5,333
Lut (5)	100%	19	**100%**	1	100%	6	100%	66	100%	3	100%	9
Mutex (7)	43%	22	29%	49	43%	71	43%	6	43%	1	**100%**	100
Qshifter (6)	33%	59	17%	0	33%	29	33%	2,103	100%	29	**100%**	8
S (52)	8%	9	8%	9	10%	452	8%	9	31%	1,538	**37%**	1,538
Szymanski (12)	67%	2,741	0%	0	25%	199	0%	0	25%	109	**75%**	4,680
TOILET (8)	75%	528	75%	84	63%	325	100%	621	**100%**	3	**100%**	3
toilet (38)	84%	47	97%	221	100%	90	97%	3,061	100%	243	**100%**	50
Tree (14)	100%	296	100%	8	100%	1	93%	1,251	**100%**	0	**100%**	0
Summary	58%	36,793	55%	29,772	56%	35,281	57%	29,518	69%	19,108	81%	23,895

Table 2: Experiments from Table 1 repeated except that the other solvers are supplied with instances preprocessed by binary clause reasoning. Again unsolved instances consumed 5,000 sec., and for each family the solver with highest success rate is show in bold, where ties are broken by time required to solve these instances. The summary line shows the average success rate over all benchmark families and the total time taken (on solved instances only).

search? To answer this question we compare the performance of 2clsQ with the other solvers on *preprocessed* instances. By using the preprocessed instances, 2clsQ's only "advantage" over the other solvers is its dynamic application of binary clause reasoning. Our results are shown in Table 2.

These results show that a significant part of the gains achieved from binary clause reasoning occurs statically prior to search. In terms of average success rate, 2clsQ still at 58% is now closer in performance to the other search based solvers all of which have gained, and still inferior to Quantor and Skizzo which have gained significantly from binary clause preprocessing. We also see that two of the families where 2clsQ was achieving superior performance, Chain and Connect, have been so reduced by preprocessing that all solvers now achieve similar performance on them. In fact, all instances of Connect are completely solved by preprocessing, and all instances of Chain are reduced to simple SAT problems by preprocessing.

Nevertheless, the results do show that dynamic binary clause reasoning improves the efficiency of search in QBF solvers. In particular, 2clsQ remains more effective than other other purely search based solvers even when the effect of inference prior to search is factored out. The question now is whether or not these improvements to search are useful, given the effectiveness of variable elimination used by Quantor and Skizzo.

4.2 Filtering out instances best solved by variable elimination

To address this question we look more closely at how effective dynamic binary clause reasoning is on instances that are more suitably solved by search. In particular, it does not really matter much if (dynamic) binary clause reasoning improves the efficiency of solving by search instances that are more easily solved by variable elimination.

We examined those instances that would be solved very quickly by variable elimination, and to factor out the effect of binary clause reasoning prior to search we first preprocessed these instances. In particular, we found that a large number of instances (approximately 285) could be solved by Quantor after preprocessing in 25 seconds or less. In fact Quantor and Skizzo are obtaining a significant head start in their average success rate over the search base solvers from these "easy" instances.

After filtering out these instances a number of benchmark families were completely eliminated. That is, all of their instances were best suited for variable elimination after preprocessing. This left us with the benchmark families Adder, adder, C, Connect, Counter, EV-Pursue, K, Mutex, S, Toilet and Szymanski. However, even among these families several instances were eliminated as being easy. In this analysis we also eliminated all instances that could not be solved by any of the solvers as such instances are not useful when comparing solvers. In total we ended up with 72 instances remaining in 10 different benchmark families.

Table 3: Solver performance on "non-easy" preprocessed instances (i.e., instances that could not be solved in 25 seconds by Quantor after preprocessing. Uniquely solved instances shown in bold.

Family	*Instance*	*2clsQ*	*Quaffle*	*Qube*	*SQBF*	*Quantor*	*Skizzo*
ADDER	*Adder2-4-c*	0	-	26	-	-	111
	Adder2-6-c	7	-	-	-	-	-
	Adder2-8-s	-	-	-	-	-	12
	Adder2-8-c	16	-	-	-	-	-
	Adder2-10-s	-	-	-	-	-	437
	Adder2-10-c	3,812	-	-	-	-	-
	Adder2-12-s	-	-	-	-	-	230
	Adder2-12-c	1,432	-	-	-	-	-
adder	***adder-8-sat***	-	-	-	-	-	12
	adder-8-unsat	-	0	0	0	-	-
	adder-10-unsat	-	0	0	935	-	-
	adder-12-sat	-	-	-	-	-	314
	adder-12-unsat	-	0	0	191	-	-
	adder-14-unsat	-	0	0	419	-	-
	adder-16-unsat	0	2	0	-	-	-
C	***C6288-10-1-1-out***	-	-	-	-	-	1,436
	C880-10-1-1-inp	1	4	3	3	905	23
Counter	***counter-16***	-	-	-	-	-	721
	counter-r-8	-	-	-	-	60	1

Continued on next page

Table 3—continued from previous page

Family	*Instance*	*2clsQ*	*Quaffle*	*Qube*	*SQBF*	*Quantor*	*Skizzo*
	counter-re-8	-	-	-	-	79	3
EV-Pursue	*ev-pr-4x4-5-3-1-lg*	1	1	0	1	82	24
	ev-pr-4x4-5-3-1-s	-	-	-	-	-	6
	ev-pr-4x4-7-3-1-lg	17	3	16	1	-	1,469
	ev-pr-4x4-7-3-1-s	-	-	-	-	-	973
	ev-pr-4x4-9-3-1-lg	180	65	2,174	2	-	-
	ev-pr-4x4-9-3-1-s	-	-	-	-	-	1,679
	ev-pr-4x4-11-3-1-lg	390	990	-	3	-	-
	ev-pr-4x4-13-3-1-lg	-	-	-	4	-	-
	ev-pr-4x4-15-3-1-lg	-	-	-	5	-	-
	ev-pr-4x4-17-3-1-lg	-	-	-	7	-	-
	ev-pr-6x6-5-5-1-2-lg	4	5	2	24	-	2
	ev-pr-6x6-5-5-1-2-s	-	-	-	-	-	258
	ev-pr-6x6-7-5-1-2-lg	60	67	44	172	-	2
	ev-pr-6x6-7-5-1-2-s	-	-	-	-	-	462
	ev-pr-6x6-9-5-1-2-lg	823	784	-	3,708	-	235
	ev-pr-6x6-11-5-1-2-s	-	-	-	-	-	606
	ev-pr-8x8-5-7-1-2-lg	3	2	3	2	4,727	3
	ev-pr-8x8-7-7-1-2-lg	68	9	298	578	-	8
	ev-pr-8x8-9-7-1-2-lg	1,292	34	-	-	-	12
	ev-pr-8x8-11-7-1-2-lg	-	-	-	-	-	18
K	*k-branch-n-4*	141	-	93	1,190	-	12
	k-branch-n-8	-	-	-	-	-	40
	k-branch-p-4	1,858	389	20	147	32	0
	k-branch-p-8	-	-	-	-	-	0
	k-branch-p-12	-	-	-	-	-	52
	k-d4-n-8	-	-	-	-	-	0
	k-d4-n-12	-	-	-	-	-	0
	k-d4-n-16	-	-	-	-	-	0
	k-d4-n-20	-	-	-	-	-	1
	k-d4-n-21	-	-	-	-	-	1
	k-lin-n-20	1,493	-	1,370	-	66	74
	k-lin-n-21	1,511	-	1,593	-	82	87
	k-ph-n-16	287	261	4,729	4,334	198	198
	k-ph-n-20	2,636	2,204	-	-	1,790	1,806
	k-ph-n-21	4,254	3,668	-	-	2,950	2,977
	k-ph-p-12	-	-	-	-	1,689	-
Mutex	***mutex-16s***	-	-	-	-	-	1
	mutex-32s	-	-	-	-	-	9
	mutex-64s	-	-	-	-	-	22
	mutex-128s	-	-	-	-	-	70
S	*s499-d4-s*	-	-	-	-	228	107

Continued on next page

Table 3—continued from previous page							
Family	*Instance*	*2clsQ*	*Quaffle*	*Qube*	*SQBF*	*Quantor*	*Skizzo*
	s499-d8-s	-	-	-	-	-	1,878
	s641-d2-s	-	-	-	-	294	18
	s713-d2-s	-	-	-	-	448	29
	s820-d2-s	-	-	-	-	429	33
	s3330-d2-s	-	-	-	-	107	11
Szymanski	*szymanski-12-s*	221	-	-	-	105	1,183
	szymanski-14-s	677	-	-	-	-	954
	szymanski-16-s	1,780	-	-	-	-	1,992
	szymanski-18-s	-	-	-	-	-	373
Toilet	*toilet-a-10-01.16*	-	59	37	-	103	32
	toilet-c-10-01.16	-	72	46	655	110	9
Solved Instances		27	21	22	20	20	57
Total time on solved instances		23,238	11,884	10,628	13,019	14,534	21,252
Number of Uniquely solved Instances		3	0	0	3	1	22

In Table 3 we show the results of the solvers on these remaining preprocessed instances. In the table a '-' is used to indicate that the particular solver could not solve the instances within a 5,000 CPU second time bound. These results show that dynamic binary clause reasoning as performed in 2clsQ is effective on these harder instances. 2clsQ solves more of these instances than any other solver (27) except Skizzo. We also see that Quantor (i.e., pure variable elimination with 20 solved instances is less effective on these remaining instances than the improved search achieved by dynamic binary clause reasoning in 2clsQ. We also see that Skizzo, with its combination of search and variable elimination remains by far the most effective approach on these remaining instances with 57 instances solved.

Finally, if we look at the number of uniquely solved instances we see that both 2clsQ and SQBF can solve 3 instances not solvable by any other solver. These include instances that to the best our knowledge have never been solved before, e.g.,'Adder2-10-c' and 'Adder2-12-c'. These two solver embed techniques for improving search, and we see that these techniques can be useful for improving our ability to solve QBF. Skizzo can 20 instances not solvable by any other solver, so we see that combining variable elimination and search appears to be the most powerful current technique for solving QBF. However, the search employed by Skizzo does not include the innovations of SQBF or 2clsQ. Hence, our results point to at least one direction for building a QBF solver superior to any that currently exists.

It is also worth noting that 2clsQ and SQBF implement many of the techniques of Quaffle and Qube, so it is hardly surprising that all of the instances solved by these two solvers are also solved by some other search based solver. This does not detract from the techniques pioneered in these solvers, like clause and cube learning, which are essential for search based solvers. The uniquely solved instances speaks instead to the value of the new techniques utilized in other solvers: variable elimination in Quantor, binary clause reasoning in 2clsQ, SAT solving lookahead in SQBF, and the mixture of variable elimination and search in Skizzo. The data indicates that these new techniques all have some value in improving our ability to solve QBF.

5 Conclusion

Our main conclusion is that extended binary clause reasoning is effective for QBF. If used prior to search in a preprocessor it is able to speed up both search based and variable elimination based solvers, as shown in [4]. Our empirical results also show that such reasoning can also be useful in a dynamic context, and that certain problem instances can be solved with such reasoning that do not seem to be otherwise solvable.

However, although our empirical results identify binary clause reasoning as being useful techniques for solving QBF, understanding more clearly how to best to combine this reasoning with other kinds of inference, especially variable elimination, remains an open question. In future work we plan to investigate this question more fully to see if we can find ways of applying binary clause reasoning in a more focused manner that can cooperate with other kinds of inference.

References

1. Van Gelder, A., Tsuji, Y.K.: Satisfiability testing with more reasoning and less guessing. In Johnson, D., Trick, M., eds.: Cliques, Coloring and Satisfiability. Volume 26 of DIMACS Series. American Mathematical Society (1996) 559–586
2. Bacchus, F.: Enhancing davis putnam with extended binary clause reasoning. In: Eighteenth national conference on Artificial intelligence. (2002) 613–619
3. Bacchus, F., Winter, J.: Effective preprocessing with hyper-resolution and equality reduction. In: Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT 2003), Lecture Notes in Computer Science 2919. (2003) 341–355
4. Samulowitz, H., Davies, J., Bacchus, F.: Preprocessing QBF. Submitted to CP (2006)
5. Büning, H.K., Karpinski, M., Flügel, A.: Resolution for quantified boolean formulas. Inf. Comput. **117** (1995) 12–18
6. Zhang, L., Madigan, C.F., Moskewicz, M.W., Malik, S.: Efficient conflict driven learning in a Boolean satisfiability solver. In: International Conference on Computer-Aided Design (ICCAD'01). (2001) 279–285
7. Zhang, L., Malik, S.: Towards symmetric treatment of conflicts and satisfaction in quantified boolean satisfiability solver. In: Principles and Practice of Constraint Programming (CP2002). (2002) 185–199
8. Giunchiglia, E., Narizzano, M., Tacchella, A.: Quantified Boolean Formulas satisfiability library (QBFLIB) (2001) http://www.qbflib.org/.
9. Samulowitz, H., Bacchus, F.: QBF Solver 2clsQ (2006) available at http://www.cs.toronto.edu/~fbacchus/sat.html.
10. Biere, A.: Resolve and expand. In: Seventh International Conference on Theory and Applications of Satisfiability Testing (SAT). (2004) 238–246
11. Giunchiglia, E., Narizzano, M., Tacchella, A.: QUBE: A system for deciding quantified boolean formulas satisfiability. In: International Joint Conference on Automated Reasoning (IJCAR). (2001) 364–369
12. Benedetti, M.: skizzo: a QBF decision procedure based on propositional skolemization and symbolic reasoning. Technical Report TR04-11-03 (2004)
13. Samulowitz, H., Bacchus, F.: Using SAT in QBF. In: Principles and Practice of Constraint Programming, Springer-Verlag, New York (2005) 578–592 available at http://www.cs.toronto.edu/~fbacchus/sat.html.