Using SAT in QBF*

Horst Samulowitz and Fahiem Bacchus
Department of Computer Science, University of Toronto, Canada.
[horst|fbacchus]@cs.toronto.edu

May 1, 2006

Abstract

QBF is the problem of deciding the satis£ability of quanti£ed boolean formulae in which variables can be either universally or existentially quanti£ed. QBF generalizes SAT and is in practice much harder to solve than SAT. One of the sources of added complexity in QBF arises from the restrictions quanti£er nesting places on the variable orderings that can be utilized during backtracking search. In this paper we present a technique for alleviating some of this complexity by utilizing an order unconstrained SAT solver during QBF solving. The innovation of our paper lies in the integration of SAT and QBF. We have developed methods that allow information obtained from each solver to be used to improve the performance of the other. Unlike previous attempts to avoid the ordering constraints imposed by quanti£er nesting, our algorithm retains the polynomial space requirements of standard backtracking search. Our empirical results demonstrate that our techniques allow improvements over the current state-of-the-art in QBF solvers.

1. Introduction

QBF is the problem of deciding the satis£ability of a quanti£ed boolean formula where variables can be either universally or existentially quanti£ed. It generalizes SAT in which all variables are (implicitly) existentially quanti£ed. Adding universally quanti£ed variables yields a considerable increase in expressive power, and consequently QBF and QCSPs can compactly represent a much wider range of problems than SAT and ordinary CSPs. These include problems like conditional planning, non-monotonic reasoning, problems in electronic design automation, scheduling, model checking and veri£cation, see, e.g.,(Egly et al., 2000). However, this added expressiveness comes with a price. Namely QBF is much more dif£cult to solve than SAT. From the point of view of complexity theory QBF is PSPACE-complete where as SAT is "only" NP-complete (Stockmeyer & Meyer, 1973). Despite this intrinsically high complexity the goal of developing practically useful QBF solvers still seems to be feasible given suf£cient conceptual and technical advances. This paper presents some new techniques that make progress towards this goal.

Most current QBF solvers, e.g., QuBE (Giunchiglia et al., 2001), Quaf¤e (Zhang & Malik, 2002) are adaptations of the classic DPLL backtracking search algorithm originally developed for solving SAT (Davis et al., 1962). There are two main properties of QBF that must be accommodated by the search. First, the search must solve both settings of every universal variable, and second the variable ordering followed during search must respect the ordering imposed by quanti£er nesting. Both of these properties make QBF solving slower than SAT. The £rst property is intrinsic to QBF, and must be accommodated in some fashion by any QBF solver. The second property is, however, somewhat more tractable, and various attempts have been made to avoid the variable ordering constraint. To date, however, all techniques for avoiding this constraint require exponential space in general, e.g., the Skolemization/expansion approach used by the Quantor (Biere, 2004) and Skizzo (Benedetti, 2004) solvers and the BDD technique used in (Audemard & Saïs, 2005).

In this paper we develop an algorithm that makes extensive use of order-free SAT solving in an attempt to alleviate (but not completely remove) the variable ordering constraint. Our algorithm retains the important polynomial space property of backtracking search. It can also use any extra space that can be provided to improve its performance, but extra space is not required for correctness (this is a common feature with current SAT and QBF backtracking solvers). We utilize a backtracking SAT solver in a backtracking QBF solver. Because both solvers are doing backtracking search we are able to develop techniques to integrate them very tightly. For example, both solvers search the same tree and share all of their datastructures, including using the same stack to store the current path. The

*This work also previously appeared in CP 2005 key innovation of our method lies in techniques for sharing information between the two solvers so that information computed during SAT solving can be used to improve QBF solving and vice versa. The result is a QBF solver that is able to improve on current state of the art on a number of benchmark suites.

2. Background

A quanti£ed boolean formula has the form $\vec{Q}.F$, where F is a propositional formula expressed in CNF and $\vec{Q}$ is a sequence of quanti£ed variables ($\forall x$ or $\exists x$). We require that no variable appears twice in Q and that the set of variables in F and Q is identical. A **quanti£er block** qb of $\vec{Q}$ is a maximal subsequence of $\vec{Q}$ where every variable in qb has the same type of quanti£er. We order the quanti£er blocks by their sequence of appearance in $\vec{Q}$: $qb_1 < qb_2$ iff qb_1 appears before qb_2 in $\vec{Q}$. Each variable x in F appears in some quanti£er block, which we denote as $qb(x)$, and the ordering of the quanti£er blocks imposes a partial order on the variables. For two variables x and y we say that $x <_{qb} y$ iff $qb(x) < qb(y)$. Note that the variables in the same quanti£er block are unordered, so we write $x \leq_{qb} y$ iff $qb(x) \leq qb(y)$. We also say that x is **universal** (**existential**) if its quanti£er in $\vec{Q}$ is $\forall$ ($\exists$). A SAT model $\mathcal{M}_S$ of a CNF formula F is a truth assignment π to the variables of F that satis£es every clause in F. We denote the value of a variable v in π by $\pi(v)$. In contrast a QBF model (**Q-model**) $\mathcal{M}_Q$ of a quanti£ed formula $\vec{Q}.F$ 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 $\mathcal{M}_Q$, if n assigns a truth value to a universal variable x then n has exactly one sibling that assigns the opposite truth value to x, and if n assigns a truth value to an existential variable then n has no siblings. For every sequence of truth assignments π from the root to a leaf of $\mathcal{M}_Q$ we have: (2) π must assign the variables in an order that respects $<_{qb}$. That is if n assigns x and one of n's ancestors assigns y then we must have that $y \leq_{qb} x$. And (3) π is a SAT model of F. A Q-model has a path for every possible setting of the universal variables of $\vec{Q}$, and thus has size exponential in the number of universals contained in $\vec{Q}$. We say that a QBF $\vec{Q}.F$ is QSAT if it has a Q-model. The QBF problem is to determine whether or not $\vec{Q}.F$ is QSAT.

DPLL works on the principle of assigning variables, simplifying the formula to account for that assignment and then recursively solving the simpli£ed formula. The **reduction** of a formula $\vec{Q}.F$ by a literal ℓ (denoted by $\vec{Q}.F|_\ell$) is the new formula $\vec{Q}'.F'$ where F' is F with all clauses containing ℓ marked as being satis£ed (implicitly removed) and $\neg\ell$ marked as falsi£ed in all remaining clauses (implicitly ℓ has been removed from these clauses), and $\vec{Q}'$ is $\vec{Q}$ with the variable of ℓ and its quanti£er removed. For example, $\big(\forall xz.\exists y.(\neg y, x, z) \wedge (\neg x, y)\big)|_{\neg x} = \forall z.\exists y(\neg y, z)$, where $(\neg x, y)$ has been marked as satis£ed and x has been marked as falsi£ed in $(\neg y, x, z)$. An alternative view of conditions (2) and (3) on a Q-model given above is that the subtree below every node n must be a Q-model of the formula $\vec{Q}.F|_{\pi_n}$ where π_n is the sequence of literals made true on the path from the root to (and including) n. From the de£nition of a Q-model it follows that if F' is logically equivalent to F (F' has the same SAT models as F) then $\vec{Q}.F$ is QSAT if and only if $\vec{Q}.F'$ is QSAT: condition 3 above is invariant for F and F'. Thus unit propagation and clause learning can be performed without changing $\vec{Q}.F$'s QSAT status: both of these transform F to a logically equivalent F'. A QSAT preserving (but not SAT preserving) transformation that can additionally be performed on $\vec{Q}.F$ is **universal reduction**. The universal reduction of a clause c is to remove all universal variables v from c such that for every other variable x in c we have $x \leq_{qb} v$. Such universals are called tailing. The intuition is as follows. Say that $v \in c$ is a tailing universal, then in any Q-Model, c must be satis£ed along any path prior to v being instantiated. (Thus c with v removed imposes the same constraint on the set of Q-models as does c). If not then since v is universal, any path that fails to satisfy c prior to instantiating v must have an extension in which v is set to false: but then that extension will falsify c and violate condition (3). We call the application of unit propagation and universal reduction until closure **Q-propagation**, and denote by $QProp(\vec{Q}.F)$ the new formula that results from Q-propagation. In Q-propagation any universal reduction steps are always performed prior to any unit propagation steps: a unit clause containing only a universal variable should yield the empty clause rather than forcing the universal. The algorithm utilized in modern SAT solvers (e.g., (Moskewicz et al., 2001)) can be adapted to solve QBF. A recursive version of this algorithm is shown in Fig. 1.

Modern backtracking QBF solvers employ two nonchronological backtracking schemes: con¤ict analysis and solution analysis. Con¤ict analysis is a standard SAT technique that involves learning new clauses via a resolution process. A failure deadend (line 2) is reached when F contains a clause in which all literals have been falsi£ed by some subset of the literals that reduced F at the previous levels (the pre£x). From this falsi£ed clause a new falsi£ed clause c can be learned via a process of resolution and universal reduction (con¤ict analysis). DPLL-QBF will then backtrack to the **asserting** level of c, which is the level where all but one of the literals in c have been falsi£ed. This is the level where c is made unit (line 4). After returning from all levels deeper than *BTLevel* (line 13-14 or 19-20), the solver arrives at line 12 or line 19, where we now have that the new clause c is unit and forces ℓ. Notice

1: $\langle$bool *Result*, literal *forced*, int *BTLevel*$\rangle$ **QBF-DPLL**($\vec{Q}.F$, *Level*)
2: **if** F *contains a falsi£ed clause* **then**
3:     *Compute new clause* c *by Con¤ict Analysis*
4:     *forced* = deepest literal in c and *BTLevel* = level c is made unit
5:     **return** $\langle$*FAIL*, *forced*, *BTLevel*$\rangle$
6: **if** *all clauses of* F *are satis£ed* **then**
7:     *Compute Backtrack Level (BTLevel) by Solution (Cube) Analysis*
8:     **return** $\langle$*SUCCEED*, –, *BTLevel*$\rangle$
9: Pick v from the £rst quanti£er block and let $\ell = v$ or $\neg v$
10: **repeat**
11:     $\vec{Q}.F = QProp\big(\vec{Q}.F|_\ell\big)$
12:     $\langle$*Result*, ℓ, *BTLevel*$\rangle$ = **QBF-DPLL**$\big(\vec{Q}.F, Level+1\big)$
13:     **if** *BTLevel* $<$ *Level* **then**
14:         **return** $\langle$*Result*, ℓ, *BTLevel*$\rangle$
15: **until** *Result* == *SUCCEED*   /* v *must be universal for this to happen* */
16: let ℓ be v's opposite value from line 9.
17: **repeat**
18:     $\vec{Q}.F = QProp\big(\vec{Q}.F|_\ell\big)$
19:     $\langle$*Result*, ℓ, *BTLevel*$\rangle$ = **QBF-DPLL**$\big(\vec{Q}.F, Level+1\big)$
20:     **if** *BTLevel* $<$ *Level* **then**
21:         **return** $\langle$*Result*, ℓ, *BTLevel*$\rangle$
22: **until** TRUE   /* *line 19 will eventually return BTLevel* $<$ *Level* */

Figure 1. DPLL for QBF

that the solver does not actually undo the original decision made at this level (the setting of the variable v chosen at line 9). Rather it simply augments the reduction of $\vec{Q}.F$ by the new unit implicant ℓ (line 11 and 18). Thus the solver might return to this level on failure a number of times: each time it discovers that another literal is implied at this level. Eventually, the recursive call at line 12 returns success at this level or returns to a higher level. (Each failure return sets another variable, so a failure return to this level at line 12 can only occur a £nite number of times.) Success returns occur as a consequence of solution analysis (line 7).

Solution analysis is a technique particular to QBF that identi£es a subset of the assignments that are suf£cient to make the QBF QSAT. This subset of assignments is called a cube. The solver can then backtrack to the deepest universal in the cube, skipping other universals not mentioned in the cube and any existentials irrespective of whether or not they are in the cube. Thus line 16 (success return) can be reached only if v is universal. A cube containing one setting of a universal can be combined with another cube containing the other setting to obtain a new cube in a cube resolution process akin to the resolution of clauses. In particular, if the deepest universal in the cube has already had its other value solved, the solver will combine these two cubes and remove the deepest universal. Hence, on success the solver always backtracks to a universal variable whose other side is not yet solved (line 12), and thus the recursive call on line 19 can never return with a successful result. We can, however, return from the call at line 19 a number of times with newly implied literals learned from failures by con¤ict analysis.

At line 9 we see that QBF-DPLL must always branch on a variable from the outermost quanti£er block. This imposes a constraint on the possible variable orderings the search can use. We now turn to a new algorithm S-QBF that tries to alleviate this constraint on variable ordering imposed by the quanti£er pre£x $\vec{Q}$.

3. S-QBF

As explained in the introduction there is no escaping the fact that in QBF we have to ensure that both settings of each universal variable are solvable. The constraint on variable ordering imposed by the quanti£er sequencing can also be a signi£cant impediment to performance. In SAT, e.g., it is provable that an in¤exible variable ordering can cause an exponential explosion in the size of the backtracking search tree. That is, there exist families of UNSAT problems for which **any** DPLL search tree where each branch follows a £xed variable ordering is exponential in size, whereas a quasi-polynomially ($O(n^{\log n})$) sized DPLL search tree exists when a dynamic ordering is used (Buresh-Oppenheim & Pitassi, 2003; Beame et al., 2004).This observation (also bolstered by empirical observations of the tremendous impact variable ordering has on DPLL SAT search), is the underlying motivation for our approach. In particular, consider a QBF formula $\vec{Q}.F$ in which the body F is UNSAT. If all of quanti£er blocks have size 1, QBF-DPLL will be forced to follow a £xed static variable ordering in proving $\vec{Q}.F$ to be UNQSAT. On the other hand an order unrestricted SAT solver might be able to determine that F is UNSAT very quickly, which will immediately tell us that $\vec{Q}.F$ is UNQSAT. The idea of testing the body of the formula, F, can be used recursively at every invocation of QBF-DPLL, just before line 9 prior to recursively solving the entire formula (body plus quanti£er) with the order constrained QBF search. If the body F is UNSAT, we can backtrack immediately. If F is SAT, then we still do not know whether or not $\vec{Q}.F$ is QSAT, so we have to continue recursively solving $\vec{Q}.F$ with our QBF solver.

Furthermore, if F is SAT our SAT solver will £nd a satisfying truth assignment for F. This truth assignment is a sensible candidate for the left-most path in a Q-model. So after we obtain the SAT solution we can follow this solution in the QBF solver during its £rst (left-most) descent. It can, however, be the case that the SAT truth assignment is not in fact a feasible left-most path for the QBF solver. In particular, this truth assignment might not survive the stronger

1: $\langle$bool *Result*, literal *forced*, int *BTLevel*$\rangle$ **S-QBF**($\vec{Q}.F$, *Level*, π)
2: **if** F contains a falsi£ed clause or if all of its clauses are satis£ed. **then**
3: Perform non-chronological backtracking using con¤ict or solution analysis as in QBF-DPLL lines 2-8.
4: **while** π == $\{\}$ **do** /* *No current SAT solution* */
5: $\langle\pi$, ℓ, *BTLevel*$\rangle$ = **SAT**$\big(F$, *Level*$\big)$
6: **if** *BTLevel* $<$ *Level* **then** /* ***SAT*** *can cause* ***S-QBF*** *to backtrack* */
7: **return** $\langle$*FAIL*, ℓ, *BTLevel*$\rangle$
8: $\vec{Q}.F = QProp\big(\vec{Q}.F|_{\ell}\big)$
9: Pick v from the £rst quanti£er block and let $\ell = \pi(v)$
10: **repeat** /* *Second and subsequent invocations of* ***S-QBF*** *need to £nd new SAT solution* */
11: $\vec{Q}.F = QProp\big(\vec{Q}.F|_{\ell}\big)$
12: $\langle$*Result*, ℓ, *BTLevel*$\rangle$ = **S-QBF**$\big(\vec{Q}.F, Level + 1, \pi\big)$
13: **if** *BTLevel* $<$ *Level* **then**
14: **return** $\langle$*Result*, ℓ, *BTLevel*$\rangle$
15: π = $\{\}$
16: **until** *Result* == *SUCCEED*
17: let ℓ be v's opposite value from line 9.
18: **repeat** /* *First and all subsequent invocations of* ***S-QBF*** *need to £nd new SAT solution* */
19: $\vec{Q}.F = QProp\big(\vec{Q}.F|_{\ell}\big)$
20: $\langle$*Result*, ℓ, *BTLevel*$\rangle$ = **S-QBF**$\big(\vec{Q}.F, Level + 1, \{\}\big)$
21: **if** *BTLevel* $<$ *Level* **then**
22: **return** $\langle$*Result*, ℓ, *BTLevel*$\rangle$
23: **until** TRUE /* *line 20 will eventually return BTLevel* $<$ *Level* */

Figure 2. S-QBF

Q-propagation performed by the QBF solver. Putting these pieces together we obtain the S-QBF algorithm given in Fig. 2. The algorithm is a modi£cation of QBF-DPLL. S-QBF is £rst invoked with the input formula $\vec{Q}.F$, *Level* equal to 1, and $\pi = \{\}$. Its £rst task is to £nd a SAT solution (line 4-8). The SAT solver might discover a number of literals implied at higher levels. Literals implied at higher levels cause S-QBF to backtrack, assert those literals, and then proceed downwards again. The SAT solver might also discover literals implied at the current level. These literals are used to reduce the input formula $\vec{Q}.F$ (line 8) via Q-propagation: these literals are independent of any choices made by the SAT solver so their consequences need to be accounted for by the QBF solver. After Q-propagating these implied literals the SAT solver is called again to see if it can £nd a SAT solution in light of these added constraints on F. Eventually, the SAT solver £nds a SAT solution (π is returned containing this solution), or causes a backtrack to a higher level in the QBF solver. If a solution is found, the QBF solver *heuristically* tries to follow this solution (in quanti£er order) by choosing a value for v that agrees with π (line 9). The SAT solution π is passed down to the next recursion where it is followed as far as possible, either to a failure or a Q-solution at line 2-3. Anycon¤icts encountered will cause a backtrack which will return to line 20 or 12 of some invocation after which the next invocation will call the SAT solver again. Thus the SAT solver is being used to refute UNSAT subtrees, and more importantly to compute new con¤ict clauses that can (a) cause the QBF solver to backtrack and (b) discover that various literals are implied at previous levels of the search. All of this information, computed by the SAT solver, is sound for the QBF solver: UNSAT subtrees are UNQSAT, any new clause learned by the SAT solver is a valid new clause for the QBF solver, and if a literal ℓ is SAT implied at a previous level of the tree then ℓ is Q-SAT implied at that level as well. It should be noted that the SAT solver can also make an S-QBF invocation backtrack from line 20, even though we know that the other side of the universal branched on in that invocation has already been successfully solved. This might seem strange, since at this point we already know that the current pre£x (above the *Level* of this invocation) contains at least one satisfying assignment below it. Thus one might think that the SAT solver could never then conclude that the pre£x is contradictory. However, although the pre£x is not SAT contradictory, it could still be QBF contradictory. For example, say that the pre£x contains the literal a, the body F contains the clauses $(\neg a, \neg b, c, d)$, $(\neg a, \neg b, c, \neg d)$, $(\neg a, \neg b, \neg c, d)$, $(\neg a, \neg b, \neg c, \neg d)$, b is universal, $b <_{qb} c$, and $b <_{qb} d$. The QBF solver will be able to solve the setting $\neg b$ without dif£culty, as this setting satis£es all of these clauses. However, when at line 20 the setting b is made these four clauses become contradictory. Q-propagation cannot detect the contradiction so the SAT solver will be invoked in the next recursive S-QBF call. SAT will be able to learn the new clause $(\neg a, \neg b)$, which after universal reduction becomes $(\neg a)$. This will cause the QBF solver to backtrack all the way to the point where a was added to the pre£x.

3.1. Integration of SAT and QBF.

In our implementation of S-QBF we built our own SAT solver (utilizing all of the modern techniques like 1-UIP clause learning, watched literals, etc. (Moskewicz et al., 2001)). In this way we were able to obtain a much tighter integration between the SAT solver and the QBF solver, e.g., sharing of datastructures. Clause learning is the basic unit of communication between the two solvers. As pointed out above, learned clauses are not necessary for correctness, but they are very helpful for ef£ciency. In particular, both the QBF solver, via contradictions generated via Q-propagation, and the SAT solver via contradictions generated via unit propagation can learn clauses. Universal reduction is applied to these learned clauses to make them more powerful. All of these learned clauses arise from sequences of Q-resolution steps, thus as shown in (Büning et al., 1995) they are all logical consequences of the input QBF. That is, they do not alter the QSAT status of the input. This means that any clause learned by either solver can be used by both solvers to prune paths from the search space they explore. This is useful as each solver is able to learn different kinds of clauses. In particular, since the SAT solver is order unrestricted it can learn powerful clauses via its VSIDS heuristic which would never be learned by the order restricted QBF solver. These clauses can signi£cantly prune the set of paths explored by the QBF solver. On the other hand the QBF solver is able to employ stronger Q-propagation and so it also can learn clauses that the SAT solver could never learn. These clauses allow the SAT solver to prune paths that are £ne from the point of view of SAT but which are contradictory with respect to QBF.

4. Empirical Results

We compared an implementation of our approach with two state of the art search based QBF solvers—Quaf¤e (Zhang & Malik, 2002) (version as of Feb. 2005) and Qube (release 1.3) (Giunchiglia et al., 2001). We also ran experiments with the non search based solver Quantor (Biere, 2004) (version as of Jan 2004). Like these solvers our implementation also utilizes techniques for detecting monotone literals, heuristics for guiding cube resolution, and some other standard improvements over the basic algorithm given in Fig.2. We used the following benchmark families from QBFLib: Adder, FlipFlop, VonNeumann, Counter, Toilet c/g, Robots_D2, Term, Comp, Z4ml, S1169, S1196, S298 and all instances provided by Pan and Rintanen (≈ 350 instances). In addition, we used a benchmark family introduced in (Remshagen & Truemper, to be published) called Game (120 instances). We excluded the families Mutex, Szymanski and Tree since all of them can be trivially solved by simple preprocessing. Further details will be discussed in a subsequent paper. We also excluded all of the other families from QBFLib (2004), e.g., Jmc and Uclid, because only one or two of their instances could be solved by any of the search based solvers. Due to space limitations we exclude results on any instance that had one of the following properties: (1) the difference in solving time between all search based solvers is small (less than either 200 seconds or within 10% of the fastest time); or (2) no search based solver can solve it in under 5,000 seconds. The remaining results are shown in Table 2. All experiments were performed on a 2.4 GHz Pentium IV with 3GB of RAM.

A summary of these results is presented in Table 1. In this table we show the total time used by each solver for all instances in each benchmark family (among those instances shown in Table 2). The "Total" column show the sum of the time over all benchmarks. To obtain a time in the presence of failures we added a penalty of $5,000$ seconds per failure. (Thus the times should be used only for qualitative comparisons). In addition, the table shows the percentage of failed instances for each benchmark family and in total.

Table 1 shows that our new approach improves the current state of the art in search based solvers, in aggregate solving the most problems and taking the least time of any of the solvers. S-QBF is not always the fastest solver, but it does improve on Quaf¤e and Qube on 21 out of the 68 problems reported on in Table 2. In many of the other cases it is very competitive, being the worst solver of the three search based solvers on only 9 of the 68 problems. As noted above we experimented with many other benchmarks, but on these the solvers could not be effectively discriminated. To obtain a more accurate assessment of the bene£t provided speci£cally by our new techniques for using SAT (vs. differences in implementation and heuristics), we built a derivative of S-QBF. This derivative (denoted S^-) used the same code base, the same variable ordering heuristic, the same cube learning and clause learning techniques, etc. S^- is simply S-QBF without the SAT solver. This provided us with a much more accurate control against which to assess our new techniques. The summary performance of S^-, shown in Table 1, demonstrates that although our base QBF solver is quite effective, our new techniques for using SAT yield clear performance advantages. Table 2 shows in more detail the time taken by the different solvers on individual problems (columns *S^-*, *S-QBF*, *Quaf¤e*, and *QuBE*). It is also useful to examine the effect SAT has on the size of the QBF search tree. Columns *SAT-dec*, *Q-dec*, *S^- Q-dec* of Table 2 show the number of decisions made by the SAT solver, the number of decisions made by the QBF solver (in S-QBF), and the number of decisions made by S^- (where SAT is not used). In most cases we see that the SAT solver is able to signi£cantly reduce the number of decisions the QBF solver needs to make (comparing columns *Q-dec* and *S^- Q-dec*). In fact, in many cases the sum of the SAT and QBF decisions in S-QBF is less than the number of QBF decisions used by the pure QBF solver S^-. QBF decisions are more expensive than SAT decisions as they require extra work (e.g., triggering of cubes, detecting monotone literals, detecting the empty theory). Hence reducing the number of QBF decisions has a strong impact on the run-time (e.g, in the *Blocks*, *Game*, and *Toilet* benchmarks). In our implementation SAT decisions are made 5 to 10 times faster than QBF decisions depending on the problem instance. This means that using SAT can yield improvements even when the sum of decisions in SAT and QBF is higher than the number of decision made by pure QBF (in S^-) (e.g., the *K* benchmarks). The SAT solver can, however, sometimes be a waste of time. For example the *Chain* benchmarks contain Q-propagation implication chains under which a QBF solver will never encounter a failure. In some cases SAT solving can even be harmful, as following its solutions can be misleading. For example,

Solver	*Blocks*	*Chain*	*Comp*	*Game*	*K*	*Robots*	*Term*	*Toilet*	*Total*
S-QBF	0%	66%	25%	0%	37%	0%	0%	0%	22%
	$2,991s$	$10,493s$	$5,000s$	$1,345s$	$70,848s$	$959s$	$2,577s$	$672s$	$26h$
Qube	20%	0%	75%	57%	25%	0%	66%	50%	31%
	$10,305s$	$3,499s$	$16,030$	$39,723s$	$59,594s$	$2,373s$	$12,566s$	$11,057s$	$43h$
Quaf¤e	20%	33%	0%	71%	50%	0%	0%	25%	43%
	$5,709s$	$9,978s$	$69s$	$50,217s$	$96,251s$	$410s$	$299s$	$6,057s$	$47h$
S^-	0%	66%	50%	57%	43%	0%	0%	25%	40%
	$4,932s$	$10,439s$	$10,000s$	$42,548s$	$84,279s$	$2,400s$	$3,246s$	$9,486s$	$45h$

Table 1. Summary of results reported in Table 2. Shown are the percentage of failed runs and the CPU time used.

on *k_d4_p-6* S-QBF makes many more QBF decisions than when SAT is not used (S^-). But in the vast majority of the cases SAT is more informative than misleading.

Quantor is another state of the art QBF solver, but it is not based on backtracking search. Instead Quantor utilizes a variable elimination scheme based on the original resolution procedure of Davis-Putnam (Davis & Putnam, 1960) and an additional scheme of universal expansion. It falls into the class of worst case space exponential algorithms. Quantor's approach often superior on these benchmarks. However, its failure rate is 24% which is slightly higher than that achieved by S-QBF. Furthermore, while we expect a few more problems could be solved by S-QBF given more time, Quantor is exhausting addressable memory on most of its failures. Overall, space exponential algorithms have the disadvantage that space is a much less ¤exible resource than time. The question of whether space intensive algorithms like Quantor, Skizzo (Benedetti, 2004), or QMRES (Pan & Vardi, 2004) will eventually be the best way to solve QBF remains open. However, we are more optimistic about search based methods. In particular, the wide variance in the times achieved by search based solvers shows that there is a lot of room for improvements in heuristics.

5. Conclusions

We have presented an approach for integrating order unconstrained SAT solving within an order constrained QBF solver. By utilizing clause learning techniques, and the fact that a SAT learned clause is valid for QBF, we have been able to achieve a tight integration between the SAT solver and the QBF solver so that information computed in each part can be used to improve the performance of the other part.

References

Audemard, G., & Saïs, L. (2005). A symbolic search based approach for quanti£ed boolean formulas.

Beame, P., Kautz, H., & Sabharwal, A. (2004). Towards understanding and harnessing the potential of clause learning. *Journal of Arti£cial Intelligence Research*, *22*, 319–351.

Benedetti, M. (2004). skizzo: a qbf decision procedure based on propositional skolemization and symbolic reasoning. Technical Report TR04-11-03.

Biere, A. (2004). Resolve and expand. *Seventh International Conference on Theory and Applications of Satis£ability Testing (SAT)* (pp. 238–246).

Büning, H. K., Karpinski, M., & Flügel, A. (1995). Resolution for quanti£ed boolean formulas. *Inf. Comput.*, *117*, 12–18.

Buresh-Oppenheim, J., & Pitassi, T. (2003). The complexity of resolution re£nements. *IEEE Symposium on Logic in Computer Science* (pp. 138–147).

Cadoli, M., Giovanardi, A., & Schaerf, M. (1998). An algorithm to evaluate quanti£ed boolean formulae. *Proceedings of the AAAI National Conference (AAAI)* (pp. 262–267).

Davis, M., Logemann, G., & Loveland, D. (1962). A machine program for theorem-proving. *Communications of the ACM*, *4*, 394–397.

Davis, M., & Putnam, H. (1960). A computing procedure for quanti£cation theory. *Journal of the ACM*, *7*, 201–215.

Egly, U., Eiter, T., Tompits, H., & Woltran, S. (2000). Solving advanced reasoning tasks using quanti£ed boolean formulas. *AAAI/IAAI* (pp. 417–422).

Gent, I., Hoos, H., Rowley, A., & Smyth, K. (2003). Using stochastic local search to solve quanti£ed boolean formulae. *Principles and Practice of Constraint Programming — CP'2003* (pp. 348–362).

Giunchiglia, E., Narizzano, M., & Tacchella, A. (2001). QUBE: A system for deciding quanti£ed boolean formulas satis£ability. *International Joint Conference on Automated Reasoning (IJCAR)* (pp. 364–369).

Moskewicz, M., Madigan, C., Zhao, Y., Zhang, L., & Malik, S. (2001). Chaff: Engineering an ef£cient sat solver. *Proc. of the Design Automation Conference (DAC)*.

Pan, G., & Vardi, M. Y. (2004). Symbolic decision procedures for qbf. *Principles and Practice of Constraint Programming* (pp. 453–467). Springer-Verlag, New York.

Remshagen, A., & Truemper, K. (to be published). An effective algorithm for the futile questioning problem. *Journal of Automated Reasoning*.

Rowley, A. (2005). *Forthcoming*. Doctoral dissertation, University of St. Andrews.

Stockmeyer, L., & Meyer, A. (1973). Word problems requiring exponential time. *Journal of the ACM*, 1–9.

Zhang, L., & Malik, S. (2002). Towards symmetric treatment of con¤icts and satisfaction in quanti£ed boolean satis£ability solver. *Principles and Practice of Constraint Programming (CP2002)* (pp. 185–199).

6. Appendix

6.1. Relation to Previous work

A number of other approaches have been proposed for escaping from the ordering constraints imposed by the quanti£er pre£x. Quantor (Biere, 2004), and Skizzo (Benedetti, 2004) both employ the device of removing universal variables by adding multiple copies of their scoped existentials. (A process akin to Skolemization in £rst-order logic). Once all universals have been removed the transformed theory becomes an order unconstrained SAT theory. As our empirical results demonstrate this technique can be very effective, but in general it requires exponential space. Our empirical results also demonstrate that it is not dif£cult to £nd problems solvable by QBF-DPLL that are unsolvable by Quantor (Skizzo was not yet available for experimentation).

A more recent order unconstrained approach is based on a BDD representation of a Q-model (Audemard & Saïs, 2005). The idea here is to generate arbitrary SAT solutions with a SAT solver, adding those solutions to the BDD. The BDD will eventually collapses to TRUE if the set of added SAT solutions suf£ce to form all paths in a Q-model. However, the BDD can grow to an exponential size prior to collapsing. Furthermore, the SAT solver can generate SAT solutions that form paths in disjoint Q-models—thus the BDD might be even larger as it has to represent multiple distinct Q-models before one collapses to a solution. The empirical results reported in (Audemard & Saïs, 2005) do not improve on the state of the art.

The idea of utilizing a SAT solver within QBF was £rst presented in (Cadoli et al., 1998). SAT solving was employed to determine trivial truth (satis£ability after removing all universals from every clause) and trivial falsity (unsatis£ability of the subset of clauses that contain only existentials) at every recursive call. Trivial truth is a very strong condition: the remaining theory can easily be QSAT even though it is not trivially true. Furthermore, because a different clause set is being used, the satis£ability testing employed in trivial truth cannot be used to learn clauses for the remaining QBF search. Trivial falsity on the other hand is strictly weaker than the SAT testing we employ. Trivial falsity tests SAT on a subset of the clauses, hence whenever it reports UNSAT our SAT testing will also report UNSAT. Furthermore, our SAT testing can report UNSAT even on formulas that are not trivially false.

In more closely related work an incomplete SAT solver was used (Gent et al., 2003). If a SAT solution was found it could be heuristically followed in an attempt to reach a successful leaf in the QBF search. This is quite different from our motivation which is to refute UNSAT subtree. This requires a complete SAT solver as well as a tighter integration between the SAT and QBF solvers. Empirically the WalkQSat solver reported in (Gent et al., 2003) did not display good performance. Independently to our work (Rowley, 2005) utilized a complete SAT solver (ZChaff (Moskewicz et al., 2001)). It allows the pruning of UNSAT subtrees and the computed reason for this con¤ict is used in the QBF solver to apply backtracking. However, the integration of the two solvers is not as tight as it is in our approach. For instance, the solvers operate on two distinct representations of the formula so that except for backtracking no exchange of learned clauses takes place between the SAT and QBF solvers. Furthermore, operations like the propagation of variable (un)assignments has to be performed twice.

6.2. Formal Results.

Theorem 1 *S-QBF is sound and complete.*

A sketch of the proof is as follows. First by relating the operations performed by QBF-DPLL on failure return to Q-resolution steps (Büning et al., 1995) it can be shown that QBF-DPLL will backtrack from the *root* of the search tree with *FAIL* only if its input is Q-UNSAT. Similarly it can be show that any recursive invocation of QBF-DPLL backtracks with *SUCCESS* only if its input is QSAT. Thus QBF-DPLL is sound. That it is also complete follows from the fact that no recursive call has exactly the same pre£x of assignments as another call (after a failure a new literal is added to the pre£x, and after a success the pre£x has a different value for one of the universal variables). Since there are only a £nite number of sets of assignments, there can only be a £nite number of recursive calls, and the root QBF-DPLL invocation must eventually return (with the correct answer).

SAT in S-QBF only allows S-QBF to backtrack on failure, it does not affect success backtracking. Thus, *SUCCESS* returns continue to correctly prove QSAT. Furthermore, all operations performed by SAT during failure backtracking are sound Q-resolution steps, so S-QBF also preserves the property that it backtracks from the root with *FAIL* only if its input is Q-UNSAT. That is, S-QBF retains QBF-DPLL's soundness property.

Observation 1 *S-QBF is systematic. That is, it never revisits the same set of assignments.*

The previous argument still holds so S-QBF retains the systematic property of QBF-DPLL. This also means that S-QBF is complete.

Problem Instance	*QSAT?*	*SAT-dec*	*Q-dec*	S^- *Q-dec*	S^-	*S-QBF*	*Quaffle*	*QuBE*	*Quantor*
blocks3i.5.3	0	37779	50482	439625	32.05	4.53	158.25	453.98	0.36
blocks3i.5.4	1	47300	62403	298121	11.85	3.12	11.08	4626.19	0.38
blocks4i.6.4	0	7367	6438	19931487	3116.49	0.95	**fail**	203.99	0.31
blocks4ii.6.3	0	6087	5685	6409879	1042.46	1.1	208.19	21.02	22.63
blocks4ii.7.2	0	1804960	1444039	2860315	729.34	2981.66	312.28	**fail**	43.23
chain16v.17	1	65519	131582	131582	439.97	493.32	129.3	71.14	0.04
chain19v.20	1	-	-	-	**fail**	**fail**	4849.32	1123.53	0.07
chain20v.21	1	-	-	-	**fail**	**fail**	**fail**	2304.390	0.08
comp_1_1.0_0_o	0	3401	755	-	**fail**	0.12	1.92	**fail**	0.02
comp_1_1.0_1_o	1	0	34	34	0	0	0	1030.88	0.04
comp_1_0.2_1_o	1	0	58	58	0.01	0.01	0	**fail**	0.03
comp_1_0.2_0_o	0	-	-	-	**fail**	**fail**	67.63	**fail**	0.05
game20_20_40_2	1	3855587	4425993	2754583	260.23	440.94	**fail**	98.26	0.08
game20_25_25_1	1	4517800	2213579	-	**fail**	309.46	**fail**	369.5	**fail**
game20_25_25_2	1	2109107	1168113	-	**fail**	125.29	**fail**	2874.96	**fail**
game20_25_25_3	1	920314	413170	2027831	326.64	40.06	**fail**	1150.51	**fail**
game20_25_25_4	1	3298510	1680483	-	**fail**	222.13	**fail**	1651.43	**fail**
game20_25_50_1	1	3298510	1680483	-	**fail**	221.74	**fail**	1657.63	**fail**
game50_25_25_1	1	2452664	954186	12368548	477.79	64.22	**fail**	1869.7	**fail**
game50_25_25_3	1	188743	66888	6182150	220.99	4.13	**fail**	**fail**	**fail**
game50_25_25_4	1	72203	34183	-	**fail**	1.63	**fail**	51.48	**fail**
game100_25_25_2	1	36165	24291	-	**fail**	0.73	**fail**	**fail**	9.26
game100_25_25_3	1	32923	16184	-	**fail**	0.63	4.06	**fail**	0.04
game150_25_25_1	0	0	21	21	0	0	0	**fail**	0.01
game150_25_25_2	1	208546	175239	-	**fail**	4.22	4.34	**fail**	0.01
game150_25_25_4	1	14604	13567	41798186	1262.76	0.3	208.79	**fail**	0.01
k_branch_p-5	1	-	-	-	**fail**	**fail**	**fail**	3854.78	**fail**
k_d4_p-6	0	5542611	55260801	2005	0.42	1689.13	**fail**	837.45	1.43
k_dum_n-6	1	1876929	1639193	1692680	221.21	122.79	**fail**	117.42	0.02
k_dum_n-8	1	-	-	-	**fail**	**fail**	**fail**	2916.89	0.06
k_dum_p-11	0	-	-	-	**fail**	**fail**	871.44	1014.83	5.32
k_grz_n-9	1	366963	294974	736851	117.68	22.32	3534.32	67.06	3.86
k_grz_n-12	1	1231288	1106900	2884937	3093.12	285.7	**fail**	250.53	10.3
k_grz_n-13	1	1420342	1277434	3339392	4046.65	353.39	**fail**	253.01	11.29
k_grz_n-16	1	5110635	4232820	-	**fail**	711.97	**fail**	1253.97	32.15
k_grz_n-17	1	6310863	5229135	-	**fail**	1396.91	**fail**	1321.97	20.7
k_grz_p-10	0	-	-	-	**fail**	**fail**	**fail**	164.81	6.78
k_grz_p-14	0	-	-	-	**fail**	**fail**	**fail**	1270.28	17.19
k_grz_p-16	0	-	-	-	**fail**	**fail**	2481.57	1694.67	27.73
k_grz_p-17	0	-	-	-	**fail**	**fail**	3107.51	1922.98	21.37
k_lin_n-7	1	1836874	900248	174011	404.32	194.34	169.26	49.75	454.34
k_lin_n-14	1	4503632	2422960	-	**fail**	4030.32	2525.31	1353.86	**fail**
k_lin_n-15	1	-	-		**fail**	**fail**	3008.53	2108.53	**fail**
k_path_n-5	1	3814468	3658630	3037899	473.3	493.5	**fail**	158.02	0
k_path_n-6	1	-	-	-	**fail**	**fail**	**fail**	1514.29	0.01
k_path_p-6	0	2895489	2490412	823834	101.87	406.71	270.42	30.26	0.01
k_ph_n-15	1	-	-	4072609	3731.09	**fail**	283.51	158.02	2962.78
k_poly_n-3	1	4702368	2945933	5078474	1445.27	426.24	**fail**	151.16	0
k_poly_n-4	1	-	-	-	**fail**	**fail**	**fail**	1651.2	0
k_poly_p-7	0	0	83	83	0	0	0	**fail**	0.01
k_poly_p-8	0	0	99	99	0	0	0	**fail**	0.02
k_poly_p-10	0	0	123	123	0	0	0	**fail**	0.04
k_poly_p-11	0	0	131	131	0.01	0.01	0	**fail**	0.03
k_poly_p-12	0	0	147	147	0.01	0.01	0	**fail**	0.03
k_poly_p-14	0	0	171	171	0.01	0.01	0	**fail**	0.03
k_poly_p-17	0	0	203	203	0.01	0.01	0	**fail**	0.03
k_t4p_n-2	1	2400994	2228055	1410656	645.73	709.56	**fail**	84.11	0.02
k_t4p_p-4	0	-	-	-	**fail**	**fail**	**fail**	194.57	0.1
robots1_5_2_72.7	1	21720	3002426	313292	44.14	221.7	19.64	1385.68	**fail**
robots1_5_2_42.7	0	29395	7713081	4458791	1519.08	672.14	288.06	565.01	**fail**
robots1_5_2_61.6	0	17992	4529115	4619291	836.47	268.29	99.34	424.87	**fail**
term1_1_0.2_0_i	0	2708395	2655162	2906302	3238.12	2555.78	296.52	**fail**	**fail**
term1_1_1.0_1_o	1	129	88	722	0.03	0.02	0.06	2566.76	0.07
term1_1_1.0_0_o	0	36105	6769	7276	7.86	18.65	3.11	**fail**	1.57
toilet6.1.11	0	54468	44831	108215	48.5	22.47	9.21	307.92	0.09
toilet7.1.13	0	347166	273852	1225940	3570.54	617.92	39.76	**fail**	1.14
toilet7.1.14	1	888	1097	712183	867.72	0.32	45.65	749.85	0.02
toilet10.1.20	1	57	264	-	**fail**	0.1	**fail**	**fail**	**fail**

Table 2. Benchmark Results