Preprint from Proceedings of MBMV, Kaiserslautern, Germany, March 2012

Enhanced Integration of QBF Solving Techniques

Sven Reimer
Albert-Ludwigs-Universität Freiburg
reimer@informatik.uni-freiburg.de

Florian Pigorsch
Albert-Ludwigs-Universität Freiburg
pigorsch@informatik.uni-freiburg.de

Christoph Scholl
Albert-Ludwigs-Universität Freiburg
scholl@informatik.uni-freiburg.de

Bernd Becker
Albert-Ludwigs-Universität Freiburg
becker@informatik.uni-freiburg.de

Abstract

In this paper we present a novel QBF solving technique which is based on the integration of a *search based* (DPLL) and a *rewriting based* approach: While traversing the search space in a DPLL manner, we iteratively generate many sub-problems, which are handed over to the rewriting method one by one. Instead of just communicating back satisfiability results of the individual sub-problems, we collect as many constraints derived by the rewriting based solver as possible, and transfer them back to the search based solver. This allows not only to prune the current branch, but also to avoid the unnecessary traversal of search paths in different regions of the search tree. We also discuss heuristics to determine suitable switching points between these two methods. We present first promising results that underline the potential of our approach.

1. Introduction

Quantified Boolean Formulas (QBF) are an extension of propositional formulas obtained by adding existential and universal quantifiers. They allow for a compact encoding of many problems, e. g., from verification [1, 2, 3] and planning [4]. The decision problem for QBF is PSPACE complete [5] and is assumed to be harder to solve than the SAT problem, which is NP complete [6].

This work was partly supported by the German Research Council (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS, http://www.avacs.org/).

Many QBF solvers with different solving techniques have been developed in the last years: DPLL-based approaches [7, 8, 9], as well as several rewriting-based approaches, such as resolution and expansion [10], symbolic skolemization [11], and symbolic quantifier elimination using AIGs [12]. The last QBF solver evaluation [13] has shown that no single solution technique is superior for all classes of QBF problems. One technique works well in one class of QBF problems, while a different class needs a different approach. This observation shows that solvers combining different solving techniques are the way to go.

We propose a novel combined approach, which starts the solution process with a search based solver that traverses the search tree in the order given by the quantifier prefix. At suitable points during the traversal, the search is stopped and sub-problems are generated by projecting the QBF instance to the variable assignments induced by the current search paths. These sub-problems are then given to a rewriting based solver, which solves them and returns sub-results along with additional constraints learned about the sub-problems to the search procedure. This allows the search not only to finish the current branch and backtrack to lower levels of the tree but also to prune unexplored parts of the search tree due to additional constraints returned by rewriting. In this paper, we present adaptive heuristics that determine suitable points to switch from search to rewriting, and we explore means of deriving and extracting constraints from the runs of the rewriting based solver that support the search process. We present first promising results on QBF instances used in the QBF evaluation that demonstrate the potential of our approach and that most notably show that our combined approach is superior to the two single solution techniques involved.

There is some related work on integrating and combining different solution techniques in the QBF domain:

One approach [14] first runs a rewriting based solver, taking snapshots of the intermediate results of the solver, and hands over completely to search when the progress of rewriting based solving significantly degrades. There are two main differences to our new approach: (1) the order of search and rewriting is reversed, and (2) they only perform one switch and thus do not transfer knowledge gained by the secondary solver to the first.

The closest relation to this work is given in [15, 16]. The authors apply online switching policies to alternate between search and resolution [17].

There are *portfolio* approaches, e. g. the multi-solver engine AQME [18], which selects the 'best' solver for a given instance from a set of solvers by machine-learning techniques. In contrast to our approach, the different techniques are strictly separated and no information is shared when switching to a different engine.

In [19, 20] a combined technique for propositional SAT formulas has been presented. The authors combine SAT and BDDs by decomposing the formula with DPLL and hand over sub-problems to BDDs [21]. However this approach until now in this form is not applicable to formulas with quantifiers.

The remainder of this paper is structured as follows: In Section 2 we give a short introduction to QBF. Furthermore, we outline the algorithmic structure of search based and rewriting based QBF solving. In Section 3 we present the details of our integration. In Section 4 we present experimental results obtained by our prototype implementation. Finally, we summarize our results in Section 5 and briefly discuss future work.

2. Preliminaries

2.1. Quantified Boolean Formulas

A *quantified Boolean formula* $\psi = \mathcal{Q}_1 X_1 \ldots \mathcal{Q}_n X_n . \phi(X)$ in *prenex conjunctive normal form* (*PCNF)* is composed of a *prefix* $\mathcal{Q}_1 X_1 \ldots \mathcal{Q}_n X_n$, where $\mathcal{Q}_i \in \{\exists, \forall\}$ are existential and universal quantifiers, $X_1, \ldots, X_n$ are pairwise disjoint and non-empty sets of Boolean *variables*, and the *matrix* $\phi(X)$ is a propositional formula over the set of variables $X = \bigcup_{i=1}^{n} X_i$ represented as a CNF. A CNF is a conjunction of *clauses* and a clause is a disjunction of *literals*. A literal l is a variable either in positive or negative ($\bar{l}$) occurrence. Instead of l and $\bar{l}$ we also write l^0 and l^1 respectively. For the ease of presentation, we will consider clauses as sets of literals. W.l.o.g., we restrict the prefix to a sequence of *alternating* quantifiers, such that $\mathcal{Q}_i \neq \mathcal{Q}_{i+1},\ \forall i \in \{1, \ldots, n-1\}$. The *quantification level* of a variable $x \in X_i$ is implicitly given by i.

Consider a QBF ψ, its matrix ϕ, a variable $v \in X_i,\ i \in \{1, \ldots n\}$ and its corresponding literal l. A QBF ψ_l is a formula whose prefix results from eliminating the variable v: $\mathcal{Q}_1 X_1 \ldots \mathcal{Q}_i (X_i \setminus v) \ldots \mathcal{Q}_n X_n$. Furthermore any occurrence of l ($\bar{l}$) in ϕ is substituted by TRUE (FALSE) – analogous for $\psi_{\bar{l}}$, where any occurrence of l ($\bar{l}$) in ϕ is substituted by FALSE (TRUE). Consider a sequence of literals $\sigma = l_1^{k_1}, l_2^{k_2}, \ldots, l_m^{k_m}$ with $k_j \in \{0, 1\}, j \in \{1, \ldots, m\}$. A QBF ψ_σ is given as $(\ldots((\psi_{l_1^{k_1}})_{l_2^{k_2}}) \ldots)_{l_m^{k_m}}$. We write Σ for the set of all variables which are included in the sequence σ. Note, that we also allow empty sequences.

A unit clause is a clause which contains only one literal. If the literal l of the unit clause is universally quantified, the formula is unsatisfiable, since we can not satisfy the formula for both polarities of l. Otherwise l can be directly assigned to TRUE, since this is the only possible way to satisfy the unit clause.

2.2. QBF solving techniques

Our integrated approach uses a prototype search based QBF solver, based on the SAT solver ANTOM [22] and the rewriting based solver AIGsolve [12, 23]. We will review both approaches briefly in this section.

A *search* based QBF solver chooses an unassigned variable $x \in X_r$, where r is the lowest quantification level with an unassigned variable (*branch* or *decision*). Afterwards, the CNF is updated with the chosen value and its implications until a fixed point is reached (*bound* or *deduction*). If we obtain a conflict/solution from this step, a reason for unsatisfiability/satisfiability is calculated and the solver backtracks to the last decision. Every partial assignment of the literal leads to a new sub-problem ψ_σ, where σ is the assignment of all literals so far. In contrast to the SAT problem both assignments of a universally quantified literal must lead to a satisfying result, i. e. both branches of this literal have to be traversed. The whole procedure is repeated until a conflict/solution on branching level 0 is produced.
In the past many techniques have been developed improving the basic algorithm (mostly adopted from SAT), such as *non-chronological backtracking*, *pure literal detection*, *restarts*, *conflict clause / solution term minimization* (see [24] for an overview). We briefly introduce the conflict clause /

solution term generation and minimization below.
A reason for an unsatisfiable matrix is an empty clause. A constraint is generated by traversing the implication graph and performing resolution steps on the involved clauses [25]. In addition to the SAT case we have to treat carefully the universally quantified variables [26].
A reason for a satisfiable matrix is an assignment of all variables in X, such that every clause is satisfied. To improve this reason we are looking for a subset of assignments to variables in X which is necessary to satisfy all clauses in the original matrix. Therefore, we primarily try to eliminate variables with a universal quantifier from the reason, since the less universal variables occur in the reason the more is pruned from the search space. Consider a literal $x_u^{k_u}$. To eliminate this literal from the solution term we look at any clause which contains $x_u^{k_u}$, searching for another literal that also satisfies the clause.
In both cases we are able to extract a new constraint from the generated and minimized reason, which prunes the search space. These constraints are finally added to the QBF ψ.

A *rewriting* based QBF solver eliminates variables until the formula (respectively the symbolic representation of the formula) is trivially unsatisfiable/satisfiable. We will give a brief introduction to the symbolic quantifier elimination with AIGs used in the QBF solver AIGsolve.
After preprocessing the initial QBF instance (see Section 2.3 for further details), AIGsolve scans the QBF for sets of clauses establishing functional definitions of variables, e. g. $(\overline{y} \vee x_1), \ldots, (\overline{y} \vee x_n), (y \vee \overline{x_1} \vee \ldots \vee \overline{x_n})$, which defines $y \equiv x_1 \wedge \ldots \wedge x_n$. Then a circuit-like, non-CNF representation of the QBF is created [12], instead of substituting variables (e.g. y) with definitions in the QBF by their corresponding definitions (e.g. $x_1 \wedge \ldots \wedge x_n$).
Next, quantifiers from the linear prefix are distributed into the non-CNF representation of the QBF (this construction is also called QTREE, see [14] for further details), reducing the scope of the individual quantifiers. For the internal representation of the propositional formulas, AIGsolve uses *And-Inverter Graphs* (AIGs) [27] which are Boolean circuits composed of two-input AND gates and inverters. In contrast to BDDs, AIGs are non-canonical, i. e. for each propositional formula there exist many structurally different AIG representations. This allows them to be potentially more compact than BDDs.
In its main routine, AIGsolve traverses the QTREE, removing leaves from the tree by eliminating the corresponding quantifiers of the leaf node using symbolic quantification methods on AIGs. Once all quantifiers/nodes are eliminated from the QTREE, the procedure terminates, leading either to an AIG which is constant 0 (*unsatisfiable*) or 1 (*satisfiable*). In case of a satisfiable instance with an existential outermost quantifier level, AIGsolve provides a *model* for the variables of the outermost quantification level. A literal in the model is either assigned to a certain value or marked as „don't care", i. e. this literal is not responsible for the satisfiablity of the QBF.
AIGsolve uses a sophisticated algorithm [23] for quantifier elimination, which heuristically combines cofactor-based quantifier elimination with quantification using BDDs and thus benefits from the strengths of both data structures.

2.3. Preprocessing QBF

Many QBF instances can be reduced in size of the variables and/or clauses in the CNF by more or less complex *preprocessing* steps. Many of these techniques are adopted from SAT, such as *(self)subsumption*, *equivalence reduction*, *variable elimination*, etc. In addition there are several

techniques specifically dedicated to the case of QBF, such as *universal reduction*. These techniques are described in [28] and are beyond the scope of this paper.
As a preprocessor for our front-end solver we use sQueezeBF [28], since it is one of the most powerful preprocessors for search based QBF solvers. In addition, we apply a light version of the AIGsolve preprocessor [23] for obtaining additional constraints as described in Section 3.3. This preprocessor runs on the intermediate sub-problems, which are handed over from the search based solver to the rewriting based solver. We use two different preprocessors, since AIGsolve needs dedicated methods (e. g. *gate detection*) and also suffers from some of the techniques used in sQueezeBF (e. g. variable elimination).

3. Integration

As described in the last section, a search based QBF solver decomposes the original formula into many disjunctive sub-problems ψ_σ until a conflict in the formula occurs or a solution is found for a certain assignment. One can observe that there are many QBF instances which tend to explore almost the whole search space until a conflict/solution on decision level 0 occurs.

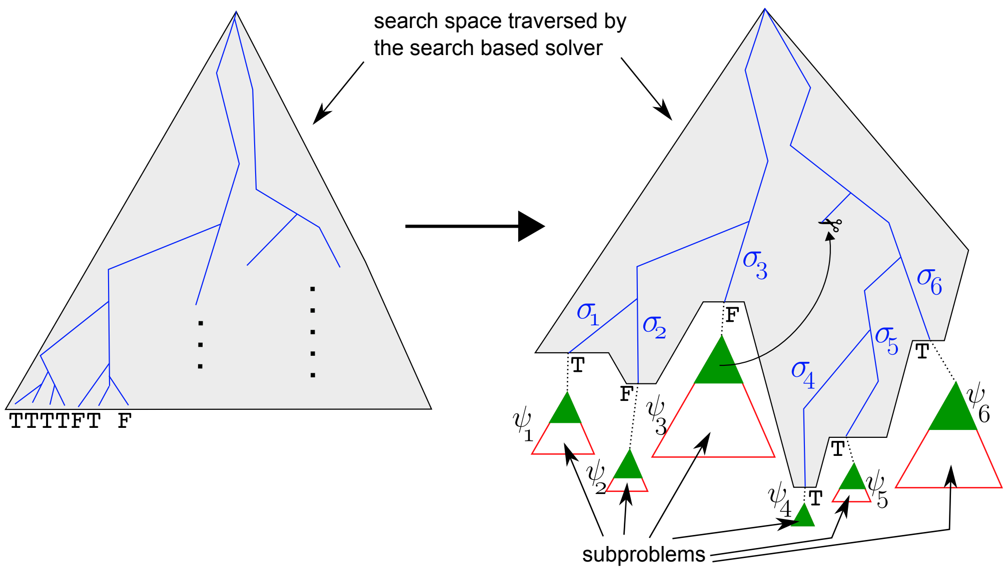

Figure 1: Decomposition into sub-problems

Figure 1 shows the basic idea of our integration. The approach is able to interrupt the search procedure and transfer the resulting sub-problem ψ_{σ_i} to the rewriting based solver AIGsolve. First these sub-problems are handed over to a preprocessor (filled triangle) producing additional constraints, which can prune paths in different parts of the search tree.
The lowest quantification level of the remaining QBF ψ_σ after a series of variable decisions is indicated by $e\,, e \geq 1$ such that $\psi_\sigma = \mathcal{Q}_e X'_e, \ldots, \mathcal{Q}_n X'_n \phi'(X'_e, \ldots X'_n)$, where $X'_i = X_i \setminus \Sigma\,, \forall i \in \{e, \ldots, n\}$. Note that – although a search based algorithm only assigns variables in order of the quantification levels – also variables from other levels can be assigned, due to the implications resulting from the bound (deduction) phase. After transferring the remaining QBF ψ_σ to AIGsolve,

AIGsolve will return an additional constraint, cutting the search space w. r. t. the current literal sequence σ.
In the following sections we present the steps of the integration, in particular: (1) how to partition the overall problem into different sub-problems, (2) how to learn from the rewriting based solver, and (3) how to gain additional constraints from the intermediate preprocessor.

3.1. Choosing suitable switching points

The longer the sequence σ the potentially easier is the sub-problem ψ_σ for AIGsolve, but the more often we have to call the back-end solver. Moreover the pruned sub-problems are quite small. If we call AIGsolve on shorter sequences, the sub-problems are more similar to the orginal instance and thus harder to solve. But if AIGsolve succeeds, we would prune large parts of the search tree. The goal is to find a trade-off between less sub-problems that are hard to solve and many sub-problems that are easy to solve.
We implement a dynamic heuristic based on the computation time t of AIGsolve for a sub-problem. Initially we call the back-end solver when all variables in X_1 are assigned. In this moment the amount of assigned variables so far $\alpha = |\Sigma|$ is stored as a reference point. We heuristically define two thresholds of the computation time t_{fast} and t_{slow}. If $t \leq t_{fast}$ ($t \geq t_{slow}$) holds, α is decreased (increased) supposing that the sub-problem is too easy (hard) for the back-end solver. From now on we switch to AIGsolve whenever α (or more) variables are assigned – α is updated after every call of AIGsolve. Additionally we define an upper limit $t_{timeout} \geq t_{slow}$. Reaching this limit, the AIGsolve instance will be terminated with an unknown result and the front-end solver resumes on current sub-problem. Apart from that, this case is treated the same way as "$t \geq t_{slow}$". In our experiments we force a strict timeout threshold (2 seconds), otherwise the back-end solver would spend to much time for potentially hard sub-problems.
Since we prefer branching heuristics that treat variables which reduce the sub-problem size as soon as possible, we choose the DCLS heuristic [29], which selects the variable that occurs most frequently in not assigned clauses. For this heuristic we focus only on the original clause set – and not the additional learned constraints – to avoid a large overhead in the calculation.
To prevent bad switching points, i. e. the value of α is over-trained to a certain part of the search space, a reset on α is performed after every internal restart of the search based solver.

3.2. Learning constraints from AIG-based solution

In this section we discuss how to learn additional constraints for the search based solver from the AIG based solver. We have to differentiate between a satisfiable and unsatisfiable sub-problem. In both cases a reason for (un)satisfiability is needed which has to be used during the backtrack procedure. For both cases there exist several approaches to improve the reason, including the one described in Section 2.2.
Existing methods for minimizing conflict clauses and solution terms are CNF-based and can not be adopted directly to AIG-based approaches, since the CNF representation of the original sub-problem is lost, when we process the sub-problem by AIGsolve. However, in the following we describe a method to derive small constraints from the back-end solver which are used for pruning the search space of the search-based solver.

First, we consider the case of a satisfiable sub-problem. The set of assignments to variables Σ, which represents the path to the sub-problem ψ_σ, provides a reason. This reason only contains variables $x \in \Sigma$, so that the related constraint is in many cases not reducible. Then we obtain a new learned constraint that only prunes *one* certain path in the search tree.
Thus, we want to improve the learned constraints. To do so, we exploit the set of satisfying assignments which is provided by AIGsolve iff $\mathcal{Q}_e = \exists$. The reason of the sub-problem can be extended by one of these satisfying assignments to variables X_e. Now the extended reason can be minimized by the method described in Section 2.2. Since it can be seen that the variables from X_e can be eliminated anyway from the reason after the minimization (see [24] for details), the number of variables in the resulting constraint is guaranteed to be less or equal to the number of variables in the original one.

A reason for an unsatisfiable matrix is a clause where every literal is assigned to false. Unfortunately AIGsolve is not able to provide any proof of unsatisfiability, i. e. we can not offer a more dedicated reason for an unsatisfiable sub-problem, yet. The least we can provide is a subset of Σ, in particular all literals which are assigned by a branch of the search based solver. A further minimization will be definitely a big issue for future work on this topic.

Note that we only hand over the original clause set to AIGsolve and not the learned constraints since we want to avoid a potential blow-up of AIGsolve's internal symbolic representation of the formula.

3.3. Learning constraints from CNF-based preprocessor

Before AIGsolve starts the main solving routine we are using a preprocessor to simplify the given sub-problem. This is motivated by two observations: $a.)$ the main AIGsolve routine benefits from the simplification of the formula w. r. t. the computation time, $b.)$ the preprocessor is able to learn additional constraints. These constraints can be used to prune different parts of the search space.

Consider a QBF ψ and a sequence of assignments σ. We have to transfer the remaining sub-problem to a preprocessor, which provides the techniques mentioned in Section 2.3. For each new learned clause ξ in the preprocessor, we memorize the set of clauses C_ξ, which are required to produce this clause. Moreover, for each clause c_σ in ψ_σ we consider the subset of assignments from σ which are needed to turn the original clause c in ψ into c_σ. This subset of assignments can be viewed as a set λ_{c_σ} of literals. We define $\lambda_\xi = \bigcup_{c_\sigma \in C_\xi} \lambda_{c_\sigma}$. Now we know the following: Assuming the assignments from λ_ξ we can derive the clause ξ from the original clauses in ψ, i.e., we can learn the constraint $\bigwedge_{l \in \lambda_\xi} l \Rightarrow \xi$ or equivalently the clause $c_{new} := \bigvee_{l \in \lambda_\xi} \bar{l} \vee \xi$.
If we kept every learned clause, we would learn many redundant constraints, which will slow down the search process. So, instead of transferring every c_{new}, we just focus on the constraints with $|\xi| = 1$, i. e. unit clauses of the preprocessor.

Example 1 *Consider a QBF* $\psi = \exists a \forall b \exists c\ d\ e\ f\ .\{a, c, \bar{d}\} \wedge \{\bar{b}, \bar{c}, d\} \wedge \{e, c, f\} \wedge \{e, \bar{d}\}$. *Let* $\sigma = \bar{a}, b, \bar{f}$ *a sequence of assignments, done by a search based solver. Then, we will obtain as* ψ_σ*:*

$$\exists a \forall b \exists c\ d\ e\ f\ .\{a, c, \bar{d}\}_{\bar{a},b,\bar{f}} \wedge \{\bar{b}, \bar{c}, d\}_{\bar{a},b,\bar{f}} \wedge \{e, c, f\}_{\bar{a},b,\bar{f}} \wedge \{e, \bar{d}\}_{\bar{a},b,\bar{f}} =$$

$$\exists c\ d\ e\ .\{c, \bar{d}\} \wedge \{\bar{c}, d\} \wedge \{e, c\} \wedge \{e, \bar{d}\}$$

We denote $\{c, \bar{d}\}$ *by* c_1*,* $\{\bar{c}, d\}$ *by* c_2*,* $\{e, c\}$ *by* c_3*, and* $\{e, \bar{d}\}$ *by* c_4*.*
Assume, that we switch to AIGsolve in this moment, starting with preprocessing the formula. First of all, we compute for every clause the set of literals, which are responsible for the clause, namely: $\lambda_{c_1} = \{\bar{a}\}$, $\lambda_{c_2} = \{b\}$, $\lambda_{c_3} = \{\bar{f}\}$, $\lambda_{c_4} = \emptyset$*. The* equivalence reduction *routine will identify that* $c \equiv d$ *holds (from* c_1 *and* c_2*), resulting in a matrix where every occurrence of* d *is replaced by* c *(in addition* c_1 *and* c_2 *will be deleted). Note, that this equivalence does not hold for* ψ*, but for* ψ_σ*. We obtain the following QBF:*

$$\exists c\ e\ .\{e, c\} \wedge \{e, \bar{c}\}$$

For $c_5 := \{e, \bar{c}\}$ *we have* $C_{c_5} = \{c_1, c_2\}$ *and* $\lambda_{c_5} = \lambda_{c_1} \cup \lambda_{c_2} = \{\bar{a}, b\}$*.*
Consider q-resolution *as the next preprocessing step. This method will resolve* c_3 *and* c_5 *to* $c_6 := \{e\}$ *with* $C_{c_6} = \{c_1, c_2, c_3\}$*, and* $\lambda_{c_6} = \lambda_{c_1} \cup \lambda_{c_2} \cup \lambda_{c_3} = \{\bar{a}, b, \bar{f}\}$*. At this moment we have learned a unit clause within the preprocessor, which will be transferred back to the search based front-end. The resulting constraint is* $c_{new} = \bigwedge_{l \in \lambda_{c_6}} l \Rightarrow c_6 = \bigvee_{l \in \lambda_{c_6}} \bar{l} \vee c_6 = \{a, \bar{b}, f, e\}$*. This constraint can be interpreted as follows: Whenever we have a sequence of assignments with* $\sigma = \bar{a}, b, \bar{f}$ *we are allowed to assign* e *with TRUE.*

4. Experimental results

We have developed a prototype implementation for our approach. The experiments were run on a Intel Core 2 Quad with 2.66 GHz and 8 GB of memory. We used a memory limit of 4 GB and a time limit of 1200 CPU seconds. We focus on 4 benchmark classes from the QBFLIB [30] problem suite.

Table 1 shows the result of a few selected benchmarks from the *mqm-Umbrella* family:

	rewriting		search			integration				
benchmark	result	time	branches	result	time	branches	sc.	cons.	result	time
Umbrella_tbm_14.tex.moduleQ2.1S.000812	unknown	TIMEOUT	1 685 099	UNSAT	321.29	134 300	21	86	UNSAT	35.37
Umbrella_tbm_21.tex.module.000029	unknown	TIMEOUT	1 006 878	unknown	TIMEOUT	234 854	55	5	UNSAT	199.97
Umbrella_tbm_23.tex.moduleQ1.2S.000001	unknown	TIMEOUT	924 719	SAT	110.76	253 252	151	409	SAT	782.03
Umbrella_tbm_25.tex.moduleQ3.2S.000063	unknown	TIMEOUT	2 180 296	unknown	TIMEOUT	253 952	21	57	UNSAT	72.53
Umbrella_tbm_26.tex.module.000004	unknown	TIMEOUT	94 415	UNSAT	14.52	40	1	0	UNSAT	13.74
Umbrella_tbm_26.tex.moduleQ3.2S.000014	unknown	TIMEOUT	1 166 263	unknown	TIMEOUT	260 609	26	5	UNSAT	151.82
Umbrella_tbm_29.tex.module.000078	SAT	20.35	749 406	unknown	TIMEOUT	265 607	86	3	SAT	217.30

Table 1: mqm-Umbrella family

In this table we compare the search based and rewriting based solver, which are involved in the integration with our approach. For all solvers we indicate the result and the runtime in seconds. For the search based solver we added the number of branches and for the integrated approach we also indicate the number of successful AIGsolve calls (*sc.*), i. e. the runtime t is below our given $t_{timeout}$ and the number of learned constraints (*cons.*), which are produced by the constraint learning routine from Section 3.3. Note, that according to Section 3.2 every result of the back-end solver produces also a constraint, which prunes the related sub-problem.
One can observe that the number of branches is significantly decreased using our approach. On the other hand the back-end solver produces an overhead of additional computation time. In most

cases this overhead is compensated by the reduction of the traversed search space. We are also able to learn constraints from the rewriting based solver, which can be reused later in the search process.

In Table 2 we give an overview for the 4 benchmark families we used:

		rewrite		search		integration	
family	#	solved	time	solved	time	solved	time
mqm-Core	63	3	20.34	26	12.56	26	12.68
mqm-Umbrella	73	19	18.40	28	16.47	36	13.30
nusmv.guidance	34	17	5.94	1	11.07	9	8.38
scholl-becker	64	50	4.74	34	10.19	38	9.53
total	234	89	49.49	89	50.29	109	43.93

Table 2: Results for different benchmark families

In this table we show for the search based solver, AIGsolve and its integrated approach for every family how many instances are solved and the amount of time (in hours) was needed. One can see that we are able to conserve the strength of the front-end solver in general. The integration is able to solve additional benchmarks, which are more dedicated for a rewriting approach using the results of the sub-problems as well as benchmarks which are hard to solve for both solvers alone (see also Table 1).

However, the results also show, that our approach is not competitive in all cases. There are several promising ideas for improvement, which are outlined in Section 5. However, the first result indicate, that our approach works in general and has a high potential.

5. Conclusions and future work

In this paper, we presented an integration of two orthogonal QBF solving techniques, namely searching and rewriting. We show that this approach reduces the number of branches made by DPLL algorithm and therewith the solution space that has to be traversed.
As future work we consider to look for better decision heuristics for the switching point between the two solution methods. To do this we investigate in machine learning approaches for automatically determined switching point.
There is also the need for tuning the involved solvers w. r. t. the integration, e. g. incremental usage of AIGsolve and its preprocessor. We also investigate a even more integrated approach in the case AIGsolve fails for a certain sub-problem. If so we may hand over the remaining sub-problem to another instance of a search based solver to preserve the work AIGsolve has done so far.
We will also investigate in further mechanism to obtain constraints, both from AIGsolve and the preprocessor. In this context we will analyze how many constraints we want to obtain from the back-end solver in general and how adding (some of) the learned constraints from the front-end can be beneficial for the solving process of the back-end solver.

Acknowledgement
The authors like to thank Paolo Marin for the helpful discussions and suggestions.

References

[1] C. Scholl and B. Becker, “Checking Equivalence for Partial Implementations,” in *Proc. of DAC 2001.*

[2] N. Dershowitz, Z. Hanna, and J. Katz, "Bounded Model Checking with QBF," in *Proc. of SAT 2005*.

[3] T. Jussila and A. Biere, "Compressing BMC Encodings with QBF," *Electr. Notes Theor. Comput. Sci.*, vol. 174, no. 3, pp. 45–56, 2007.

[4] J. Rintanen, "Constructing Conditional Plans by a Theorem-Prover," *J. Artif. Intell. Res. (JAIR)*, vol. 10, pp. 323–352, 1999.

[5] L. J. Stockmeyer and A. R. Meyer, "Word problems requiring exponential time (Preliminary Report)," in *Proceedings of the Fifth Annual ACM Symposium on Theory of Computing*, STOC '73, (New York, NY, USA), pp. 1–9, ACM, 1973.

[6] S. A. Cook, "The Complexity of Theorem-Proving Procedures," in *STOC*, pp. 151–158, ACM, 1971.

[7] E. Giunchiglia, M. Narizzano, and A. Tacchella, "QUBE: A System for Deciding Quantified Boolean Formulas Satisfiability," in *Proc. of IJCAR 2001*.

[8] L. Zhang and S. Malik, "Towards Symmetric Treatment of Conflicts And Satisfaction in Quantified Boolean Satisfiability Solver," in *Proc. of CP 2002*.

[9] F. Lonsing and A. Biere, "Integrating Dependency Schemes in Search-Based QBF Solvers," in *SAT*, pp. 158–171, 2010.

[10] A. Biere, "Resolve and Expand," in *Proc. of SAT 2004, Selected Papers*.

[11] M. Benedetti, "sKizzo: A Suite to Evaluate and Certify QBFs," in *Proc. of CADE 2005*.

[12] F. Pigorsch and C. Scholl, "Exploiting Structure in an AIG Based QBF Solver," in *Conf. on Design, Automation and Test in Europe*, April 2009.

[13] C. Peschiera, L. Pulina, A. Tacchella, U. Bubeck, O. Kullmann, and I. Lynce, "The Seventh QBF Solvers Evaluation (QBFEVAL'10)," in *Theory and Applications of Satisfiability Testing - SAT 2010*, 2010.

[14] S. Reimer, F. Pigorsch, C. Scholl, and B. Becker, "Integration of orthogonal QBF solving techniques," in *Conf. on Design, Automation and Test in Europe*, 2011.

[15] L. Pulina and A. Tacchella, "Learning to Integrate Deduction and Search in Reasoning about Quantified Boolean Formulas," in *FroCos*, 2009.

[16] L. Pulina and A. Tacchella, "A Structural Approach to Reasoning with Quantified Boolean Formulas," in *IJCAI*, pp. 596–602, 2009.

[17] H. K. Büning, M. Karpinski, and A. Flögel, "Resolution for Quantified Boolean Formulas," *Inf. Comput.*, vol. 117, no. 1, pp. 12–18, 1995.

[18] L. Pulina and A. Tacchella, "A self-adaptive multi-engine solver for quantified Boolean formulas," *Constraints*, no. 1, pp. 80–116, 2009.

[19] A. Gupta and P. Ashar, "Integrating a Boolean Satisfiability Checker and BDDs for Combinational Equivalence Checking," in *VLSI Design*, pp. 222–225, 1998.

[20] A. Gupta, Z. Yang, P. Ashar, L. Zhang, and S. Malik, "Partition-Based Decision Heuristics for Image Computation Using SAT and BDDs," in *ICCAD*, pp. 286–292, 2001.

[21] R. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," *IEEE Trans. on Comp.*, vol. 35, no. 8, pp. 677–691, 1986.

[22] M. L. Tobias Schubert and B. Becker, "Antom | solver description," in *SAT Race*, 2010.

[23] F. Pigorsch and C. Scholl, "An AIG-Based QBF-Solver Using SAT for Preprocessing," in *Proceedings of Design Automation Conference (DAC '10)*, pp. 170–175, June 2010.

[24] E. Giunchiglia, M. Narizzano, and A. Tacchella, "Clause/Term Resolution and Learning in the Evaluation of Quantified Boolean Formulas," *J. Artif. Intell. Res. (JAIR)*, vol. 26, pp. 371–416, 2006.

[25] L. Zhang, C. F. Madigan, M. W. Moskewicz, and S. Malik, "Efficient Conflict Driven Learning in Boolean Satisfiability Solver," in *ICCAD*, pp. 279–285, 2001.

[26] L. Zhang and S. Malik, "Conflict Driven Learning in a Quantified Boolean Satisfiability Solver," in *Proc. of ICCAD 2002*.

[27] A. Kuehlmann, M. K. Ganai, and V. Paruthi, "Circuit-based Boolean Reasoning," in *Proc. of DAC 2001*.

[28] E. Giunchiglia, P. Marin, and M. Narizzano, "sQueezBF: An Effective Preprocessor for QBF," in *Proc. of QiCP 2008*.

[29] J. P. M. Silva, "The Impact of Branching Heuristics in Propositional Satisfiability Algorithms," in *EPIA*, pp. 62–74, 1999.

[30] C. Peschiera, L. Pulina, and A. Tacchella, "QBFLIB." `http://www.qbflib.org`.