# Integration of Orthogonal QBF Solving Techniques

Sven Reimer, Florian Pigorsch, Christoph Scholl, and Bernd Becker
Albert-Ludwigs-Universität Freiburg, Institut für Informatik,
D-79110 Freiburg im Breisgau, Germany
{reimer, pigorsch, scholl, becker}@informatik.uni-freiburg.de

***Abstract*—In this paper we present a method for integrating two complementary solving techniques for QBF formulas, i. e. *variable elimination* based on an AIG-framework and *search* with DPLL based solving. We develop a sophisticated mechanism for coupling these techniques, enabling the transfer of partial results from the variable elimination part to the search part. This includes the definition of heuristics to (1) determine appropriate points in time to snapshot the current partial result during variable elimination (by estimating its quality) and (2) switch from variable elimination to search-based methods (applied to the best known snapshot) when the progress of variable elimination is supposed to be too slow or when representation sizes grow too fast. We will show in the experimental section that our combined approach is clearly superior to both individual methods run in a stand-alone manner. Moreover, our combined approach significantly outperforms all other state-of-the-art solvers.**

## I. Introduction

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

In recent years a large number of powerful QBF solvers with different solving techniques have been developed: search-based approaches [7], [8], [9] which extend the DPLL algorithm known from SAT [10], as well as different solvers based on eliminating variables, such as resolution and expansion [11], symbolic skolemization [12], and symbolic quantifier elimination using AIGs [13].

The recent QBF solver evaluation [14] has shown that until now there is no dominant solution technique being superior for all classes of QBF problems. When looking at single classes of QBF instances, one can observe that the performance of current QBF solvers often is class specific: one technique works well in one class, while a different class needs a different approach. This observation indicates that a combination of different solution techniques may be beneficial.

Several approaches for combining different QBF solving techniques have recently been proposed: The multi-solver engine AQME [15] applies a *portfolio* approach incorporating several different back-end solvers. Based on machine-learning techniques the 'best' solver for a given instance is preselected. The main drawback of AQME is the strict separation of the back-end solvers – no information is shared when switching to a different engine. Another approach has been presented in [16]. Here the authors modify a search-based solver by dynamically selecting branching heuristics, again based on machine-learning. The solvers presented in [17], [18] apply online switching policies to alternate between search and resolution [19]. The authors prove that such a dynamic combination in principle works quite well, but they fail to be competitive with state-of-the-art solvers.

In this paper, we present an integrated approach for combining orthogonal solution techniques, that especially focuses on exchanging partial solutions between the involved solvers. In particular, our approach first runs a variable elimination based solver AIGsolve [13], [20] on a QBF instance, taking snapshots of the current QBF formula and carefully observing the solver's progress. Once a degradation of the progress or a blowup in representation sizes is observed, the best known snapshot is handed over to a search-based solver, which then tries to solve the instance.

As we will show in the experimental section of this paper, our integrated approach leads to impressive results on the QBFEVAL data-sets [21], [22], clearly outperforming state-of-the-art solvers. We will also show that our way of *combining* solvers is indeed beneficial: the integration is able to solve QBFs that cannot be solved by the incorporated solvers alone.

In contrast to previous works, our method on the one hand uses a stronger coupling than AQME by sharing partial results between the solvers which allows it to perform better than each solver for its own, and on the other hand it is able to inherit the individual power of the employed solving techniques.

The paper is structured as follows: In Sect. II we give a short introduction to QBF and the representation that is used in our approach. Furthermore, we outline the algorithmic structure of AIGsolve. In Sect. III we present the details of our integration. In particular, we define criteria to decide whether to abort AIGsolve, and present quality measures for snapshots of the current solver state. In Sect. IV we discuss the experimental results obtained by our prototype implementation, comparing our approach to other approaches for combining solvers as well as other state-of-the-art solvers. Finally, we summarize our results in Sect. V and briefly discuss future work.

## II. Preliminaries

### A. Quantified Boolean Formulas

A quantified boolean formula $\psi$ in *prenex conjunctive normal form* (*PCNF*) is a formula $\mathcal{Q}_1 x_1 \ldots \mathcal{Q}_n x_n . \phi(x_1, \ldots, x_n)$ where $\mathcal{Q}_i \in \{\exists, \forall\}$ are existential and universal quantifiers and $\phi(x_1, \ldots, x_n)$ is a propositional formula over the boolean variables $x_1, \ldots, x_n$ represented as a CNF. $\mathcal{Q}_1 x_1 \ldots \mathcal{Q}_n x_n$ is called the prefix and $\phi$ is called the matrix of $\psi$.

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/).

978-3-9810801-7-9/DATE11/©2011 EDAA

A less restrictive form of QBFs are *quantifier trees* [23] which in contrast to PCNF allow for tree shaped quantifier structures but are still based on CNF representations, i. e. the matrix parts are in clausal form. We extend this concept to *generalized quantifier trees*, which instead of clauses use arbitrary propositional formulas for representing the matrix parts of the QBFs:

A *generalized quantifier tree* (*QTREE*) over a set of boolean variables $V$ is a tree, where each node $n$

1) is labelled with a variable $var_n \in V$ and a quantifier $\mathcal{Q}_n \in \{\exists, \forall\}$,
2) is annotated with a propositional formula $f_n$ over the variables occurring in the labels of nodes on the path from the root to the node $n$.

We additionally require that each variable occurs with a unique quantifier, on each path a variable occurs at most once, and for any path $p$ the order $\leq_p$ of variables induced by the path is a subset of a linear order $\leq$ on $V$.

The interpretation of a node $n$ as a QBF formula can be defined inductively:

$$qbf(n) := \mathcal{Q}_n var_n.(f_n \wedge qbf(c_n^1) \wedge \ldots \wedge qbf(c_n^m))$$

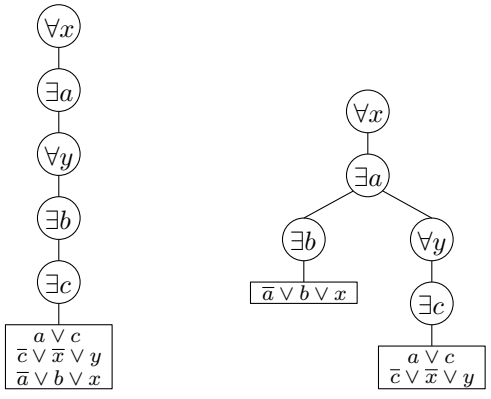
where $c_n^1, \ldots, c_n^m$ are the children of $n$.

Note that a QBF in PCNF can be trivially transformed into an equivalent QTREE, while the transformation of a QTREE into PCNF format requires a transformation of the embedded propositional formulas to CNF, which may be accomplished by *Tseitin encoding* [24] with the help of additional existential variables.

**Example 1** *Consider the following PCNF:*

$$\forall x \exists a \forall y \exists b \exists c.(a \vee c) \wedge (\overline{c} \vee \overline{x} \vee y) \wedge (\overline{a} \vee b \vee x)$$

*Fig. 1(a) shows a QTREE with linear structure directly derived from the formula. From this graph we can extract an equivalent QTREE which is shown in Fig. 1(b). Note, that each path is consistent with the linear order induced by the quantifier prefix. Instead of clauses we are allowed to use other representations of propositional formulas, e. g. AIGs, which are briefly introduced in the next section.*




(a) QTREE with linear structure (b) Equivalent QTREE

Fig. 1. Two equivalent QTREEs

### B. Variable elimination with AIGsolve

AIGsolve [13], [20] is a QBF solver for instances in PCNF format that is based on variable elimination.

After preprocessing the initial QBF instance with a number of techniques (see [20] for 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 $y$ to be equivalent to $x_1 \wedge \ldots \wedge x_n$. Instead of substituting variables (e.g. $y$) with definitions in the QBF by their corresponding definitions (e.g. $x_1 \wedge \ldots \wedge x_n$) and applying the distributive law to produce a flat PCNF representation, AIGsolve creates a circuit-like, non-CNF representation of the QBF [13]. Next, quantifiers from the linear prefix are distributed into the non-CNF representation of the QBF, reducing the scope of the individual quantifiers and producing a QTREE. For the internal representation of the QTREE, AIGsolve uses *And-Inverter Graphs* (AIGs) [25] which basically are boolean circuits composed solely of two-input AND gates and inverters, and thus can be used for representing arbitrary propositional formulas. In contrast to BDDs [26], AIGs are non-canonical – for each propositional formula there exist structurally different AIG representations – which allows them to be potentially more compact than BDDs.

In its main phase, AIGsolve traverses the QTREE in a depth-first manner, removing leaf nodes from the tree by eliminating the corresponding quantifiers using AIG operations. The procedure terminates once all nodes are removed from the QTREE and thus all quantifiers are eliminated from the QBF, leading either to an AIG which is constant 0 (*unsatisfiable*) or 1 (*satisfiable*).

For quantifier elimination, AIGsolve uses a sophisticated algorithm [20], which heuristically combines cofactor-based quantifier elimination with quantification using BDDs and thus benefits from the strengths of both data structures.

## III. Integration

### A. Motivation and Overview

It can be observed that the elimination of variables with AIGsolve may result in an exponential blow up of the AIG structure due to the cofactor-based method. In the worst case the AIG doubles with each elimination of a quantified variable and thus the evaluation of the following operations on the structure may be slowed down resulting in violation of memory or timing constraints. However, even if this worst case occurs, there are also intermediate results within this process which could be beneficial to other solvers as long as there is no extreme blowup of the AIG structure.

Consequently, in our approach we run AIGsolve as a front-end solver until it solves the formula or until we observe an undesired behavior. During the solving process we save partial results and – in contrast to pure portfolio approaches – reuse them by handing them over to a back-end solver, if we have to abort the front-end solver.

As we will show in the experimental results this approach works with different search-based back-end solvers. We are using search-based solvers since experimental results show that this technique is highly orthogonal to AIGsolve w. r. t. the solved instances.

To do so, we have to find an appropriate point in time during the solving process of AIGsolve when we expect that AIGsolve will not be able to solve the QBF instance efficiently.

Furthermore, have to save QTREEs during the solving process (*'snapshots'*) and keep them for the back-end solver.

Of course, we also need to find good moments to take such snapshots. As a naïve approach we could hand over the result in the moment the solver aborts. As we will show in our experimental results, in most cases this is not a good idea, since the resulting PCNF is harder to solve than the original instance.

In the following subsections we give more details of our approach.

### B. Translation

First of all we describe how to take a snapshot of a QTREE $t$. We have to compute the quantifier prefix of the remaining QBF formula as well as we have to consider all propositional formulas $f_n$ of $t$ represented as AIGs and convert them into a suitable format for other solvers, i. e. in PCNF.

As mentioned before, the transformation of the AIGs $f_n$ can be done with the Tseitin encoding technique by introducing propositional variables for every node in the AIG. The resulting CNF consists of three clauses per AIG node.

The prefix can be computed by traversing $t$ in a breadth-first search manner from the root node to the leaves collecting the variables $var_n$ and quantifiers $\mathcal{Q}_n$ for each node $n$ in $t$. Due to the properties of the orders $\leq_p$ of paths in $t$ this induces a linear quantifier prefix which is consistent with the original one. The additional variables from the Tseitin encoding are appended to the end of the quantifier prefix.

### C. Handover

We have to identify whether the solving process will probably be successful, and if not, we hand over an intermediate result. On the one hand we would like to notice as soon as possible when a instance cannot be solved within an adequate time, on the other hand we do not want to hand over, if AIGsolve is actually able to return a result later on.

We make two main observations:

(1) AIGsolve is suffering from large AIG structures, since quantifications and compaction methods then need much more time. If the quantification routine shows an exponential growth of AIG nodes in addition, we suppose that the solver is not able to eliminate the remaining quantifiers within the given time bounds.

(2) In some cases the mere number of quantifications cannot be processed within a given time limit although the amount of AIG nodes remains small and no exponential growth is observed.

We summarize how we heuristically cover (1) and (2) in our approach:

For (1) we define a empirically specified value indicating an upper bound for the size of the AIG structure. If the AIG size exceeds this value we start watching the increase of the AIG after each quantification. If we notice a significant blowup (i. e. doubling of the size in one step or 'almost' a doubling per step over two consecutive steps), we stop AIGsolve. For (2) we carefully monitor the process, i. e. the number of eliminated nodes of the QTREE per time unit and compare this with the remaining nodes and time. We estimate whether the remaining time will be insufficient for solving the problem (hereby, we make the assumption, that the progress will stay more or less constant).[1]

If one of these cases occurs, the AIGsolve routine will stop and hand over a snapshot to a back-end solver. We check these criteria after each elimination step.

### D. Snapshot moments

We want to determine a suitable moment for taking a snapshot of the current QTREE in a way that it is beneficial for the subsequent back-end solver. The solving process should be rewarded for eliminating (especially universally) quantified variables, since in general solving techniques benefit from fewer (universal) variables. As a result of eliminating these variables the AIG can blow up exponentially during elimination operations. This behavior has to be punished in a suitable way, given that the AIG size is directly related to the number of resulting clauses in the PCNF. We define a measure of the cost $m(t)$ of a QTREE $t$, which can basically described as follows:

$$m(t) = u\_weight(u) \cdot e\_weight(e) \cdot size$$

where $u\_weight(u)$ and $e\_weight(e)$ define weighting factors w. r. t. the number $u$ ($e$) of current universal (existential) quantifications of $t$. $size$ corresponds to the number of clauses and additional variables of the resulting PCNF due to the Tseitin encoding on the basis of the AIG nodes. Since the elimination of universally quantified variables is in general more beneficial, we choose $u\_weight$ and $e\_weight$ such that for the same parameters $u\_weight$ results in a higher factor than $e\_weight$. The particular values are based on empirical results and will be specified in the experimental results.

It is essential to do this calculation after any quantification in the solving process. Once the current QTREE has smaller costs according to our cost measure compared to the cost of the latest snapshot, we translate the current state as described in Sect. III-B and store it as the latest snapshot.

Note that we only have one snapshot at any point in time – the one which is estimated as best. As a reference point we take a first snapshot right after the preprocessing but before the solving process.

### E. Final comparison

We observe that there are QBF instances in which *any* (even the initial) snapshot is worse than the original problem for a specific back-end solver. Other solvers do not necessarily benefit from the AIGsolve preprocessor since the techniques specifically focus on AIG (but not PCNF) simplification. On the other hand, elimination of variables and quantifications may compensate this.

Thus we consider a second cost measure to heuristically determine whether a (given) search-based solver will *really* benefit from a snapshot. For this purpose we need a measure that evaluates the PCNF result for a snapshot of a QTREE and compare its 'quality' to the original PCNF.

We use a QBF preprocessor both on the taken snapshot and the original formula and compare the resulting PCNFs. For

[1] Of course, if we do not have a given time limit, the second criterion will never take effect.

the comparison we basically use the measure $m$ we defined before, but with a different definition for the value $size$. Here, that value contains the actual number of literals, i. e. the sum of all clause sizes, after QBF preprocessing. This final comparison takes place only once with the last snapshot taken by AIGsolve; in order to save runtime we refrain from performing comparisons with all snapshots taken during the run of AIGsolve.

### F. Tuning AIGsolve for the integration

We experimented with heuristics for processing the QTREE in various orders within AIGsolve. Different options for choosing the next node to process are: Choosing the node whose variable has the largest (smallest) quantification level / the node with the smallest (largest) number of AIG nodes / the node with the largest (smallest) depth within the QTREE, choosing the node according to a depth-first traversal of the QTREE, or any combination of these heuristics. We observe no significant influence of these heuristics on solving with AIGsolve alone, but it turned out that the integration benefits from several heuristics – especially from choosing a variable with the largest quantification level, because this strategy favors complete eliminations of (universally quantified) variables early in the solution process before the AIG blows up.

Furthermore, we want to decrease the number of newly introduced variables. Therefore, we take advantage of the AIG structure by detecting *multi-input* and-structures, i. e. multiple nodes that correspond to one logical AND with more than two inputs. For these structures we only have to introduce one additional variable instead of one variable per node and instead of receiving three clauses per node we get $i+1$ clauses, where $i$ is the number of inputs of the multi-input and-structure. In many cases the number of introduced variables and clauses can be reduced significantly with this technique. Of course, we have to take account of this with regard to our measure $m$ (see Sect. III-D).

### G. Algorithm

Algorithm 1 describes the basic routine of the integration technique. *extractQTREE()* generates a QTREE from a PCNF as described in Sect. II-A. *estm()* either calculates our cost measure for a QTREE or PCNF. *handoverResult()* checks the handover criteria as stated in Sect. III-C. The *eliminateNodes()* method eliminates variables by symbolic quantification on the AIG structure and *takeSnapshot()* translates the current QTREE as described in Sect. III-B. If every node of $t$ could be eliminated, we can return the result of $t$ given by the remaining AIG. Finally, *preprocess* and *solve* can be arbitrary QBF preprocessors and solvers, respectively.

### H. Example

We will demonstrate this approach on a specific benchmark from QBFEVAL'10:

The *Core1108_tbm_21.tex.module.000026* instance from the 'Kontchakov'-family [27] contains 359 (105) existentially (universally) quantified variables and 2555 clauses after preprocessing. The gray bars in Fig. 2 show the development of our cost measure $m$ within the solving process. The triangles indicate a moment when a snapshot is taken, i. e. when the current value of $m$ is smaller than the last estimated one. On the very left the initial snapshot can be seen. On the very right one can see the last snapshot that is finally made (the filled green triangle). Right after this snapshot was taken, $m$ is increasing fast (note, this diagram has a logarithmic scale) due to an exponential increase of the AIG nodes during quantification. After eliminating 330 variables we observed that our first handover criterion took effect whereupon the solution process was canceled.

```
Input: PCNF F
t ← extractQTREE(F);  lastm ← ∞;
repeat
    if estm(t) < lastm then
        takeSnapshot(t);  lastm ← estm(t);
    if handoverResult(t) then
        PCNF S ← exportSnapshot();
        break;
    eliminateNodes(t);
until nodes(t) = ∅ ;
if nodes(t) = ∅ then
    return result(t);
F' ← preprocess(F);  S' ← preprocess(S);
if estm(F') < estm(S') then
    return solve(F');
else
    return solve(S');
```

**Algorithm 1**: Integration

Altogether we took 79 snapshots requiring overall 0.11 seconds of CPU time. The last snapshot was taken after 6.62 seconds and we stopped the AIG based solving process after a total of 153.8 seconds. The last elimination steps needed most of the time because of the increasing number of AIG nodes. At the moment of the last snapshot, the number of universal quantifiers had been decreased substantially from 105 to 50. Furthermore 236 existential quantifications of the preprocessed formula had been eliminated. The PCNF that was generated from the last snapshot had 1306 (50) existential (universal) quantifiers and 4485 clauses. The increasing number of existential quantifications originates from the additional Tseitin variables introduced.[2] (This effect is already diminished by

[2]Note that, if the number of universally quantified variables is small, additional Tseitin variables at the end of the quantifier prefix are not necessarily very harmful to the search-based solution process, because after assigning all universally quantified variables during search the remaining problem turns into a SAT problem.

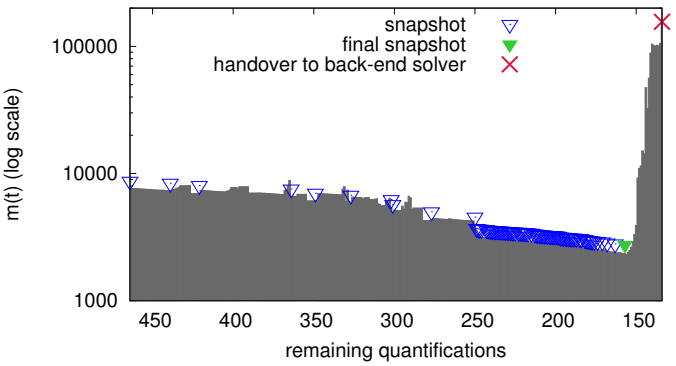



Fig. 2. Development of measure $m$

detecting multi-and structures – without this technique we would introduce 6051 clauses and 1900 (50) existentially (universally) quantified variables.) After extracting the PCNF from the last snapshot, a state-of-the-art QBF preprocessor is able to eliminate several of the new quantifications and clauses, e. g., sQueezeBF [28] is able to optimize the PCNF such that we obtain 336 (50) existentially (universally) quantified variables and 1463 clauses. Since the number of clauses, the number of existentially quantified variables, and (especially) the number of universally quantified variables was reduced, we estimate that the resulting QBF instance is easier to solve for search-based solvers than the initial problem.

## IV. Experimental Results

All experiments were run on a 16-core AMD Opteron with 2.3 GHz and 64 GB of memory. We used a time limit of 1200 CPU seconds and a memory limit of 4 GB.

As preprocessor we used sQueezeBF [28], as front-end AIGsolve [13], and as search-based back-end solvers QuBE7 [7] ('QuAIG') and DepQBF 0.1 [9] ('DepAIG'). For the experiments the parameters of measure $m(t)$ were assigned to: $u\_weight(u) = 1.01^u$ and $e\_weight(e) = 1.001^e$, where $u$ ($e$) is the number of remaining existentially (universally) quantified variables.

We compared our implementation to the QBF solvers we use in our integrated approach: AIGsolve, QuBE and DepQBF which all participated in QBFEVAL'10. In addition we used AQME-10 [15], the most successful combined solver so far (winner of QBFEVAL'10), and a portfolio based approach ('portfolio') with AIGsolve and QuBE, i. e. first AIGsolve gets 600 CPU seconds for solving the instance and if it fails, QuBE also gets 600 CPU seconds. In this approach no information is shared between the solvers.

For a first series of experiments we used the benchmark set of QBFEVAL'10 [22] consisting of 568 benchmarks. Fig. 3 shows the number of solved instances in a certain time for each solver. The total number of solved instances is indicated in the top left of the figure. One can see that AQME is able to solve many instances with a small amount of time due to its preselection technique. On the other hand, QuAIG (DepAIG) is clearly superior w. r. t. the number of solved instances for the given time: 478 (487) vs. 432. The portfolio approach solves 454 instances. All other solvers, which are based on a single solution strategy, solve less than 400 instances.

To confirm these results also for other classes of benchmarks with even more diversity we enlarged our benchmark set: we considered the QBFEVAL'08 set [21], containing 3328 instances, and merge it with the QBFEVAL'10 set. To avoid redundancy, we skipped the benchmarks that are only shuffled versions of other ones – this results in a total number of 3512 benchmarks. In addition, we evaluated an integration of AIGsolve and QuBE ('QuAIG naïv') which makes only restricted use of the snapshot technique: Only when one of our handover criteria is fulfilled, we take a snapshot and forward that partial result directly to sQueezeBF and QuBE.

The results are shown in Table I. For each solver the number of solved instances per benchmark class is specified as well as the total solving time in CPU hours. Failed instances (due to memory or time constraints) are considered to contribute the time limit of 1200 CPU seconds. For our integration tool we add the number of instances which are solved by the back-end solver alone ('org'), when AIGsolve fails to solve the instance, but we decide not to use any taken snapshot. Furthermore we give the number of solved instances which are solved by the back-end solver using a taken snapshot ('comb') and the number of additionally solved instances w. r. t. the solvers that are used in the integration ('additional').

Again the result of the portfolio variant (2923) shows the high orthogonality of the different solving techniques. Note that most instances that can be solved by one of the involved solvers need less than 600 CPU seconds, so that this result already comes near to the accumulated number of solved instances (2970).[3]

Our naïve approach solves 2658 instances and demonstrates the importance of the determination of a suitable snapshot moment. But even without this technique we can solve 179 more instances (mainly from 'bbox') than AIGsolve.

Our approach QuAIG, including all introduced techniques and using QuBE as back-end, is able to solve 2975 instances, i. e. we can solve more instances than both solvers together and 496 more than the best stand-alone solver AIGsolve within the given limits. (However, we do not reach the results of the front-end solver AIGsolve and the back-end solver QuBE in all cases – especially in the 'ev-pr' and 's_*' classes. In most of these cases AIGsolve behaves well for a long time according to our handover criteria, so that there remains only a small amount of time for running the back-end solver.)

In 46 cases the instance is solved by running the back-end solver on the original formula. In these cases our final comparison method decides to use the original formula.

Overall 462 instances can be solved by using a snapshot taken for the back-end solver – mainly in the classes 'abduction', 'bbox' and 'kontchakov' which are also the classes where QuBE is able to solve significantly more instances than AIGsolve. This shows that we are able to maintain the power of the solvers within our integration.

Moreover we can solve 96 instances – mainly in the classes 'bbox', 'kontchakov', 'sorting' and 'tipfixpoint' – which cannot be solved by the involved solvers alone within the given limits. Altogether we clearly outperform the current state-of-the-solvers as well as the portfolio-based approach of the two solvers that participated in the integration.

[3]This is the number of instances solved by AIGsolve *or* QuBE within the full time limit of 1200 CPU seconds.

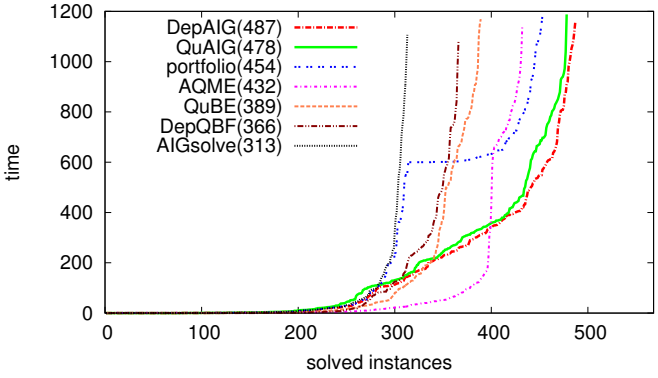



Fig. 3. Result of QBFEVAL'10

TABLE I
COMPARISON WITH OTHER SOLVERS

| | | AIGsolve | | QuBE | | DepQBF | | AQME | | QuAIG portfolio | | QuAIG naïv | | QuAIG | | | | | DepAIG | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| class | count | solved | time | solved | time | solved | time | solved | time | solved | time | solved | time | solved | time | org | comb | additional | solved | time | org | comb | additional |
| abduction | 303 | 181 | 43.7 | 287 | 7.6 | 285 | 7.9 | 268 | 12.2 | **286** | 7.8 | 212 | 36.8 | 282 | 27.2 | 9 | 93 | 1 | 272 | 26.1 | 14 | 73 | 3 |
| adder | 32 | **27** | 2.1 | 6 | 8.7 | 6 | 8.7 | 12 | 7.0 | **27** | 5.6 | **27** | 2.0 | 26 | 2.5 | 0 | 0 | 0 | **27** | 2.2 | 0 | 0 | 0 |
| bbox | 478 | 214 | 88.5 | **419** | 22.3 | 341 | 51.1 | 305 | 60.2 | 414 | 23.0 | 360 | 51.3 | 416 | 40.3 | 0 | 199 | 13 | 402 | 35.4 | 0 | 189 | 34 |
| blocks | 13 | 9 | 1.4 | 9 | 1.4 | 10 | 1.0 | **13** | 0.0 | 9 | 1.4 | 8 | 1.7 | 9 | 1.4 | 0 | 0 | 0 | 11 | 1.0 | 2 | 1 | 1 |
| bmc | 132 | 109 | 8.8 | 65 | 25.7 | 79 | 18.9 | 110 | 8.9 | **111** | 17.9 | 109 | 8.6 | 110 | 8.4 | 0 | 0 | 1 | 109 | 8.9 | 0 | 0 | 0 |
| circuits | 63 | **9** | 18.2 | 4 | 19.7 | 5 | 19.3 | 7 | 18.8 | **9** | 19.0 | **9** | 18.2 | **9** | 18.2 | 0 | 0 | 0 | **9** | 18.2 | 0 | 0 | 0 |
| counter | 24 | **16** | 3.4 | 9 | 5.1 | 10 | 4.9 | 12 | 4.0 | 13 | 4.6 | **16** | 3.4 | **16** | 3.4 | 0 | 0 | 0 | **16** | 3.4 | 0 | 0 | 0 |
| debug | 38 | 0 | 12.7 | 0 | 12.7 | 0 | 12.7 | **22** | 6.9 | 0 | 12.7 | 0 | 12.7 | 0 | 12.7 | 0 | 0 | 0 | 0 | 12.7 | 0 | 0 | 0 |
| Ev-pr | 38 | 2 | 12.0 | **21** | 6.5 | 13 | 9.2 | 15 | 7.9 | 20 | 6.6 | 4 | 11.8 | 14 | 9.9 | 11 | 1 | 0 | 14 | 9.8 | 9 | 3 | 0 |
| jmc-quant+sqr | 20 | **16** | 1.3 | 6 | 4.7 | 0 | 6.7 | 0 | 6.7 | **16** | 3.0 | **16** | 1.3 | **16** | 1.3 | 0 | 0 | 0 | **16** | 1.3 | 0 | 0 | 0 |
| k_* | 378 | 352 | 10.6 | 219 | 54.7 | 146 | 81.4 | 327 | 18.2 | 356 | 32.1 | 352 | 10.8 | **361** | 9.4 | 4 | 6 | 5 | **361** | 8.8 | 5 | 5 | 5 |
| kontchakov | 136 | 26 | 38.5 | 96 | 19.5 | **135** | 4.0 | 106 | 15.3 | 83 | 20.9 | 25 | 38.5 | 120 | 18.3 | 4 | 91 | 21 | 127 | 15.4 | 1 | 102 | 1 |
| planning | 24 | 9 | 5.2 | 8 | 5.4 | 10 | 5.0 | **14** | 3.9 | 10 | 5.2 | 10 | 5.2 | 10 | 5.0 | 1 | 0 | 0 | 11 | 5.1 | 1 | 0 | 1 |
| Scholl-becker | 64 | 50 | 4.8 | 39 | 8.7 | 38 | 9.2 | 44 | 7.5 | **51** | 6.7 | 50 | 4.8 | **51** | 4.7 | 0 | 2 | 0 | **51** | 4.8 | 0 | 2 | 1 |
| sorting | 84 | 15 | 23.1 | 36 | 16.4 | 43 | 15.9 | **61** | 8.7 | 36 | 16.4 | 25 | 20.7 | 57 | 14.2 | 9 | 34 | 29 | 44 | 16.4 | 1 | 25 | 21 |
| s_* | 171 | **102** | 28.9 | 72 | 39.2 | 1 | 56.7 | 16 | 52.4 | 98 | 36.4 | 82 | 35.4 | 80 | 37.9 | 1 | 4 | 0 | 87 | 37.1 | 0 | 3 | 2 |
| stmt | 713 | 694 | 10.5 | 618 | 32.1 | 11 | 234.1 | 307 | 136.3 | 701 | 18.7 | 686 | 13.4 | **703** | 8.0 | 4 | 1 | 7 | 698 | 10.1 | 8 | 0 | 10 |
| szymanski | 12 | **12** | 0.0 | **12** | 0.1 | 2 | 3.3 | **12** | 0.2 | **12** | 0.1 | **12** | 0.0 | **12** | 0.0 | 0 | 0 | 0 | **12** | 0.0 | 0 | 0 | 0 |
| tipdiameter | 203 | 192 | 4.4 | 162 | 15.3 | 77 | 42.6 | 150 | 18.5 | **200** | 9.4 | 194 | 3.8 | 197 | 3.5 | 0 | 7 | 1 | 195 | 4.0 | 0 | 4 | 3 |
| tipfixpoint | 446 | 315 | 45.7 | 246 | 72.5 | 71 | 125.1 | 106 | 114.7 | 337 | 57.6 | 332 | 41.5 | **357** | 34.7 | 2 | 23 | 18 | 326 | 43.0 | 3 | 1 | 21 |
| toilet | 46 | 42 | 1.3 | 43 | 1.1 | 44 | 0.7 | 45 | 0.4 | **45** | 0.8 | 42 | 1.3 | 42 | 1.3 | 0 | 0 | 0 | 42 | 1.3 | 0 | 0 | 0 |
| other | 94 | 87 | 2.6 | 85 | 3.6 | 76 | 6.9 | **89** | 1.9 | **89** | 2.8 | 87 | 2.8 | 87 | 2.8 | 1 | 1 | 0 | 86 | 3.0 | 0 | 1 | 0 |
| total | 3512 | 2479 | 367.8 | 2462 | 382.8 | 1403 | 725.4 | 2041 | 510.7 | 2923 | 308.6 | 2658 | 252.5 | **2975** | 297.3 | 46 | 462 | 96 | 2916 | 302.4 | 44 | 409 | 103 |

Using DepQBF as back-end solver we are able to solve 2916 benchmarks. The number of accumulated solved instances of the involved solvers amounts to 2913, i. e. this variant is also able to solve more instances than these solvers alone (both with their full time limit of 1200 CPU seconds).

One can observe that AQME can solve more instances than any other stand-alone solver in QBFEVAL'10, but it fails to be competitive in our second run, including QBFEVAL'08 benchmarks. AIGsolve behaves the other way around. In contrast to this, our integrating approach is robust against different benchmark sets.

## V. CONCLUSIONS

In this paper, we presented an integration of two orthogonal QBF solving techniques, namely variable elimination and search. As shown in the experiments, our approach is able to handover the partial result of the first solution technique and provide a simpler QBF problem to the back-end solver. This approach also outperforms existing portfolio-based methods as well as all other state-of-the-art solvers.

Until now, our integration is one-way in the sense that AIGsolve transfers results to a DPLL based solver. We are working on symmetric integrations where the solvers bring in their own strengths, profit from each other and jointly contribute to the solution of QBF problems.

## REFERENCES


[1] 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*, ser. STOC '73. New York, NY, USA: ACM, 1973, pp. 1–9.

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

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

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

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

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

[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*, 2010, pp. 158–171.

[10] M. Davis, G. Logemann, and D. W. Loveland, "A machine program for theorem-proving," *Commun. ACM*, vol. 5, no. 7, pp. 394–397, 1962.

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

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

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

[14] 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.

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

[16] H. Samulowitz and R. Memisevic, "Learning to Solve QBF," in *AAAI*. AAAI Press, 2007, pp. 255–260.

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

[18] ——, "A structural approach to reasoning with quantified boolean formulas," in *IJCAI*, 2009, pp. 596–602.

[19] H. K. Büning, M. Karpinski, and A. Flögel, "Resolution for quantified boolean formulas," *Inf. Comput.*, vol. 117, no. 1, pp. 12–18, 1995.

[20] F. Pigorsch and C. Scholl, "An AIG-Based QBF-Solver Using SAT for Preprocessing," in *Proceedings of the 47th Annual Design Automation Conference (DAC '10)*, June 2010, pp. 170–175.

[21] C. Peschiera, L. Pulina, and A. Tacchella, "QBF Evaluation 2008," http://www.qbfeval.org/2008.

[22] ——, "QBF Evaluation 2010," http://www.qbfeval.org/2010.

[23] M. Benedetti, "Quantifier Trees for QBFs," in *Proc. of SAT 2005*.

[24] G. Tseitin, "On the complexity of derivations in propositional calculus," in *Studies in Constructive Mathematics and Mathematical Logics*, 1968.

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

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

[27] R. Kontchakov, L. Pulina, U. Sattler, T. Schneider, P. Selmer, F. Wolter, and M. Zakharyaschev, "Minimal Module Extraction from DL-Lite Ontologies Using QBF Solvers," in *IJCAI*, 2009.

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