An AIG-Based QBF-Solver Using SAT for Preprocessing

Florian Pigorsch
Institut für Informatik
Albert-Ludwigs-Universität Freiburg, Germany
pigorsch@informatik.uni-freiburg.de

Christoph Scholl
Institut für Informatik
Albert-Ludwigs-Universität Freiburg, Germany
scholl@informatik.uni-freiburg.de

ABSTRACT

In this paper we present a solver for Quantified Boolean Formulas (QBFs) which is based on And-Inverter Graphs (AIGs). We use a new quantifier elimination method for AIGs, which heuristically combines cofactor-based quantifier elimination with quantification using BDDs and thus benefits from the strengths of both data structures. Moreover, we present a novel SAT-based method for preprocessing QBFs that is able to efficiently detect variables with forced truth assignments, allowing for an elimination of these variables from the input formula. We describe the used algorithm which heavily relies on the incremental features of modern SAT-solvers. Experimental results demonstrate that our preprocessing method can significantly improve the performance of QBF preprocessing and thus is able to accelerate the overall solving process when used in combination with state-of-the-art QBF-solvers. In particular, we integrated the preprocessing technique as well as the quantifier elimination method into the QBF-solver *AIGSolve*, allowing it to outperform state-of-the-art solvers.

Categories and Subject Descriptors: J.6 [Computer Aided Engineering]: Computer Aided Design

General Terms: Algorithms, Verification

Keywords: Quantified Boolean Formulas, Boolean Satisfiability

1. INTRODUCTION

Quantified Boolean Formulas (QBF) are a generalization of propositional formulas by adding existential as well as universal quantifiers. The addition of quantifiers on the one hand allows for the compact representation of many different problems from verification (e.g. [28, 9, 15]), planning (e.g. [25]), and other domains, but on the other hand comes at the price of increased complexity: determining the satisfiability of a QBF is a PSPACE-complete problem and thus assumed to be harder to solve than the SAT problem for propositional formulas.

The importance of the problem has given rise to the development of a number of powerful QBF-solvers which are able to tackle QBF problems originating from practical applications. Such solvers include search-based approaches [32, 13] which apply an extension of the DPLL algorithm known from SAT [7], as well as solvers based on eliminating variables by different methods, such as symbolic skolemization [3], resolution and expansion [4], and symbolic quantifier elimination using AIGs [23].

In this paper we propose a novel method for quantifier elimination which is based on the symbolic quantifier elimination using AIGs [23], but combines the advantages of AIGs and BDDs [5] by integrating both AIG and BDD techniques. The success of our quantifier elimination method relies on a close interaction between AIG and BDD techniques controlled by observing the sizes of intermediate results during the solution process.

Another improvement presented in this paper is based on preprocessing. Similar to the SAT domain, where preprocessing [10] has proven to be very effective in accelerating the solving process, various preprocessors for QBF instances have been presented (e.g. [27]), successfully supporting QBF-solvers by substantially simplifying the input formulas.

One technique used by several solvers for SAT formulas is known as *failed literal detection* [18] or as unit propagation look-ahead. Given a CNF ϕ and a variable x occurring in ϕ this method speculatively performs unit propagation for $\phi \wedge x$ and $\phi \wedge \overline{x}$. If a conflict occurs during the first (second) propagation, x ($\overline{x}$) is called a *failed literal* and ϕ is simplified by permanently assigning x to $false$ ($true$).

In [29] the authors describe a similar algorithm for the use in a decision procedure for logic programming problems. They also point out that it is not necessary to examine all literals, instead literals implied by non-failed literals may be skipped.

The detection of a failed literal x can also be achieved by recursive learning [17] with depth 0, if the assignment of x to $true$ necessarily implies conflicting assignments. By increasing the recursion depth, recursive learning is able to detect even more 'failed literals' than a pure unit propagation look-ahead. However, for reasons of efficiency the recursion depth of recursive learning is usually limited to small constants.

In this paper we present a novel method for preprocessing QBFs called *constant detection*, that lifts the idea of *failed literal detection* to the QBF domain but differs from the former in two substantial aspects:

(1) Since the QBF problem is much harder to solve than the SAT problem, we can afford using a more powerful and more expensive method for constant detection than unit propagation: in fact we apply full SAT-checks for detecting constants.

(2) In order to avoid futile checks for constants, we apply a sophisticated learning scheme, that exploits satisfying assignments returned by the SAT-solver to eliminate candidates for future constants checks.

The paper is structured as follows. In Sect. 2 we briefly review the logic of Quantified Boolean Formulas, in Sect. 3 we describe

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/).

DAC'10, June 13-18, 2010, Anaheim, California, USA

our algorithm for constant detection and in particular the learning scheme, in Sect. 4 we describe the integration of constant detection and the improved quantifier elimination scheme based on a close interaction between AIGs and BDDs into the QBF-solver *AIGSolve* [23]. In the last section we report the results for an extensive set of experiments that demonstrate the effectiveness of our approach.

2. PRELIMINARIES

A quantified boolean formula $\psi \in QBF$ in prenex form is a formula $\mathcal{Q}_1 x_1 \ldots \mathcal{Q}_n x_n.\phi(x_1, \ldots, x_n)$ where $\mathcal{Q}_i \in \{\forall, \exists\}$ are universal and existential quantifiers and $\phi(x_1, \ldots, x_n)$ is a propositional formula over the boolean variables $x_1, \ldots, x_n$.

$\mathcal{Q}_1 x_1 \ldots \mathcal{Q}_n x_n$ is called the *prefix* and ϕ is called the *matrix* of ψ. If the matrix is in conjunctive normal form, a QBF is said to be in *prenex normal form*. Existentially (universally) quantified variables are called existential (universal) variables for short. The quantifier prefix induces an order $<_q$ on the variables x_i: $x_i <_q x_j$ iff x_i occurs before x_j in the prefix.

3. CONSTANT DETECTION

Theorem 1 (Simplification by Constant Literals) *Let* $\psi = \mathcal{Q}_1 x_1 \ldots \mathcal{Q}_n x_n.\phi(x_1, \ldots, x_n)$ *be a QBF.*

$$\text{If } \phi(x_1, \ldots, x_n) \rightarrow x_i \text{ then}$$

$$\psi' = \mathcal{Q}_1 x_1 \ldots \mathcal{Q}_n x_n.\phi(x_1, \ldots, x_n) \wedge x_i \text{ is equivalent to } \psi$$

Proof: Since $(\phi \rightarrow x_i) \implies (\phi \equiv (\phi \wedge x_i))$, the matrices of ψ and ψ' are equivalent, and thus the two QBFs are equivalent as well [27]. □

Analogously, Thm. 1 holds for the case that $\phi(x_1, \ldots, x_n)$ implies $\overline{x_i}$.

If we can prove that $\phi(x_1, \ldots, x_n) \rightarrow x_i$ holds, we call x_i a *constant literal* and we are allowed to add x_i as a unit clause to the QBF ψ, yielding an equivalent QBF ψ'. If x_i is universally quantified in ψ's quantifier prefix, then ψ is unsatisfiable. If x_i is existential, the resulting QBF can be simplified by unit propagation, eventually further enabling other simplification techniques.

3.1 SAT-Based Detection of Constant Literals

A naive approach to detect all constant literals of a QBF formula is given in Alg. 1.

Algorithm 1: Naive detection of constant literals

```
Input: QBF Q_1 x_1 ... Q_n x_n.φ(x_1, ..., x_n)
Output: Set of constant literals
C := ∅;
foreach x_i ∈ {x_1, ..., x_n} do
    if φ ∧ x_i is UNSAT then
        C := C ∪ {¬x_i};
    else if φ ∧ ¬x_i is UNSAT then
        C := C ∪ {x_i};
return C
```

For each variable x_i the algorithm performs SAT checks for testing whether the variable (or its negation) is a constant literal. In the worst case (if no constant literals exist) $2 \cdot n$ SAT problems are solved.

In case an individual SAT instance is satisfiable (and thus the tested literal is not constant), the SAT-solver returns a SAT-model of the SAT formula, assigning truth values to all variables occuring in the formula. These models can be exploited to prevent (unsuccessful) SAT checks for constants: suppose the SAT formula $\phi \wedge x_i$ is satisfiable and the returned model M assigns the variable x_j (occuring in ϕ) to the value $true$. Then there is no need to check whether $\phi \wedge x_j$ is unsatisfiable since M is also a model for $\phi \wedge x_j$. This observations give rise to an improved method for the detection of constant literals as shown in Alg. 2.

The algorithm first checks whether ϕ is satisfiable and immediately returns if the instance is unsatisfiable. In case of satisfiability, the SAT-model is examined: if it assigns $true$ ($false$) to a variable x_i, we know that the literal $\overline{x_i}$ (x_i) cannot be a constant literal and we record in the set T that the literal x_i ($\overline{x_i}$) still needs to be tested to be constant true. After this initial step we perform the remaining recorded tests for constants exploiting the SAT-models of satisfiable instances to refine the set T by deleting the literals corresponding to variable assignments from it.

Altogether the algorithm performs at least $|C| + 1$ SAT-checks (ignoring the case that ϕ is unsatisfiable) and at most $n+1$ checks if either all variables are constant literals or no refinement is possible.

Algorithm 2: Improved detection of constant literals

```
Input: QBF Q_1 x_1 ... Q_n x_n.φ(x_1, ..., x_n)
Output: Set of constant literals
C := ∅; T := ∅;
if φ is SAT then
    foreach lit ∈ model(φ) do
        T := T ∪ {¬lit};
else
    return UNSAT
foreach lit ∈ T do
    if φ ∧ lit is UNSAT then
        C := C ∪ {¬lit};
    else
        foreach lit' ∈ model(φ ∧ lit) do
            T := T \ {lit'};
return C
```

The number of calls to the SAT-solver can further be reduced by modifying the solver's decision heuristics: instead of deciding variables to arbitrary values[1] during the search for a satisfying assignment to disprove that a literal is constant true it would be better, if the solver preferred to decide variables to values that have not already occurred in satisfying assignments of previous constant checks, such that the resulting satisfying assignment is more successful in ruling out following checks for constants (i.e. the last **foreach**-loop in Alg. 2 is more successful in removing candidates from the set T).

After computing the initial model of the CNF ϕ in Alg. 2, we set the preferred decision value for each variable to the negation of variable's value in the initial model[2].

As we will show in the experimental section, learning from satisfying assignments as done in Alg. 2 in combination with modifying the solver's decision heuristics is indeed able to prevent a large number of unsuccessful checks for constants, such that the average number of actually performed SAT-checks decreases to only a small percentage of the possible checks.

Moreover, note that the SAT checks performed in Alg. 2 (checking ϕ or $\phi \wedge lit$ for different literals) share almost all of their clauses. Therefore we make use of the opportunity to perform *incremental* SAT checks [30]. Thereby, the solution of a series of similar problems is accelerated to a large extent, since the solution process for one SAT instance profits from knowledge (in form of conflict clauses) obtained during the solution process for similar problems.

4. QBF SOLVING

To evaluate the performance of the algorithm presented in the previous section as well as our quantifier elimination based on AIGs and BDDs we modified the QBF-solver *AIGSolve* [23]. In this section we summarize our modifications and show how they are integrated into the flow of *AIGSolve*. Finally, we illustrate the approach by analyzing the solution process for two typical examples from

[1] In our implementation we use MiniSAT [11], which decides variables to the value $false$ by default.

[2] MiniSAT provides the function "setPolarity" for modifying the default decision values of arbitrary variables.

QBF Evaluation 2008 [22].

4.1 Preprocessing

In the main loop of the preprocessing phase unit propagation [8], subsumption checking [10], for-all reduction [27], and equivalence reduction [26] are applied until closure. Finally trivial SAT checks (omitting the quantifier prefix) are performed to check whether the matrix of the formula is unsatisfiable or a tautology [6].

The SAT-based constant detection is embedded into the main preprocessing loop as depicted in Alg. 3. Note that the time the solver is allowed to spend in constant detection is strictly limited.[3] If the time for constant detection is exceeded, constant detection is aborted and the set of constants discovered until this point is returned.

Moreover, we extended *AIGSolve*'s preprocessing routine to perform a complete expansion of the universal variables [2, 4] if only a few universal variables are remaining after preprocessing, effectively turning the QBF problem into a pure SAT instance, which is then handed over to a SAT-solver. Since expansion is only applied to instances with few universal variables, the resulting formulas are of moderate size and can efficiently be solved by SAT-solvers.

Algorithm 3: Modified preprocessor

```
Input: QBF Q in prenex normal form
Output: Simplified QBF
repeat
    Q' := Q;
    repeat
        Q'' := Q;
        UnitPropagation(Q);
        Subsumption(Q);
        ForAllReduction(Q);
        EquivalenceReduction(Q);
    until Q'' = Q ;
    C := ConstantDetection(Q);
    Q := Q ⋀_{c∈C} c;
until Q' = Q ;
if Only few universal quantifiers in Q then
    Q := ExpandUniversals(Q);
    if Matrix of Q is satisfiable then
        return SAT
    else
        return UNSAT
else
    TrivialSatisfiability(Q);
return Q
```

4.2 AIG-Based Solving with BDD Support

After preprocessing, *AIGSolve* scans the remaining QBF formula for clausal gate definitions as it is done for example in SAT preprocessing [10] or by other QBF preprocessors and solvers [4, 12] and finally eliminates the defined variables by substituting them with their definitions. Unlike other approaches, *AIGSolve* does not produce a flat CNF, which may blow up during this step, but generates a compact non-CNF, circuit-like representation, which is later directly transformed into an And-Inverter Graph (AIG) [16].

Furthermore, the linear quantifier prefix of the prenex QBF formula is dissolved by pushing the quantifiers into the non-CNF matrix, producing a tree-shaped QBF formula while minimizing the scope of quantifiers as also performed in *sKizzo* [3].

As presented in [23] the original *AIGSolve* would then transform the QBF into an AIG representation and eliminate all quantifiers in a bottom-up fashion by performing cofactoring on AIG cones (similar to the BED-based approach in [1]), compressing the individual results of quantifier elimination by BDD-sweeping [24], functional reduction [20] and DAG-aware rewriting [19] of the AIG structure.

BDD-sweeping may compress the AIG representation of a function, if a BDD of reasonably small size can be constructed for this function. If such a BDD is found, a structurally equivalent AIG is built which replaces the original AIG, if this version is smaller. Motivated by the observation already made by the authors of [23] that BDD sweeping is able to compress the AIG representation in many cases, we modified the quantifier elimination algorithm to allow for an extended exploitation of good BDD representations:

Given an AIG f and a variable x which is to be quantified, we first try to construct a BDD for the function represented by f. This BDD construction is resource limited such that the procedure is aborted in case the BDD representation blows up. If a BDD cannot be computed or the BDD is too large, we perform normal cofactor based quantifier elimination, followed by AIG-rewriting and functional reduction steps to compress the resulting AIG.

If we were able to compute a reasonably small BDD for f, we perform BDD-based quantifier elimination. Here we try not only to quantify x, but also the other variables from the same (existential or universal) quantifier block as x. The BDD based quantification is performed with a size limit and it has the advantage that it can eliminate several variables at once, if successful. If the BDD based method does not fail, we transform the result back to an AIG representation, again performing rewriting and functional reduction for compressing the result.

If the quantifier elimination was performed by AIG operations and the AIG did not grow too much due to the elimination, we stick to AIG-based quantifier elimination for the next steps and avoid computing BDDs.

4.3 Examples

Whereas pure BDD based methods have not been very successful compared to other QBF solvers [21], we have made the experience that an integration of AIG- and BDD-based methods is very beneficial.

To demonstrate the interaction between AIG- and BDD-based quantifier elimination methods, we briefly look into the solver's behavior on two benchmarks from *QBF Evaluation* 2008:

The *stmt21_4_354* instance initially contains 3112 variables distributed on two quantifier blocks, as well as 25780 clauses. AIGSolve's preprocessor slightly reduces the number of variables by 5 and the number of clauses by 45. In the remaining QBF formula, AIGSolve detects and extracts 2777 functional definitions (2744 AND-gates and 33 XOR-gates), such that only a single binary clause is left in the formula and the innermost quantifier block is reduced to only 70 existential variables. The resulting structure of the QBF is shown on the left hand side of Fig. 1. Gray and white ellipses denote universal and existential quantifier blocks, annotated with the number of quantifiers. If such an ellipse has more than one outgoing edge, then it represents also an AND operation for all outgoing edges before quantification. Sets of clauses are presented as white boxes, and the extracted gate structure is shown as a gray trapezoid.

After transforming the functional definitions (2277 gates) into an AIG representation consisting of 2414 nodes, AIGSolve starts eliminating quantifiers in a bottom-up manner. The development of AIG nodes is shown in Fig. 2 (upper part). AIGSolve first tries to compute a BDD for the AIG structure which fails due to resource limits. Therefore the solver starts to eliminate the innermost existential quantifiers using AIG-based quantifier elimination. Since the number of AIG nodes is not increasing, the solver sticks to AIG-based quantification and does not try to build BDDs for the remaining existential quantifiers. It can be observed that AIG based quantifier elimination performs well for a large number of steps. Although the AIG sizes may potentially double with each quantification of a single variable in the worst case, the sizes remain small due to the compression techniques used. After performing all existential quantifications, the solver then continues to eliminate the universal quantifiers, again using the AIG-based method. Since the number of AIG nodes actually grows during some universal quan-

[3] In our implementation we used a limit of 20 seconds.

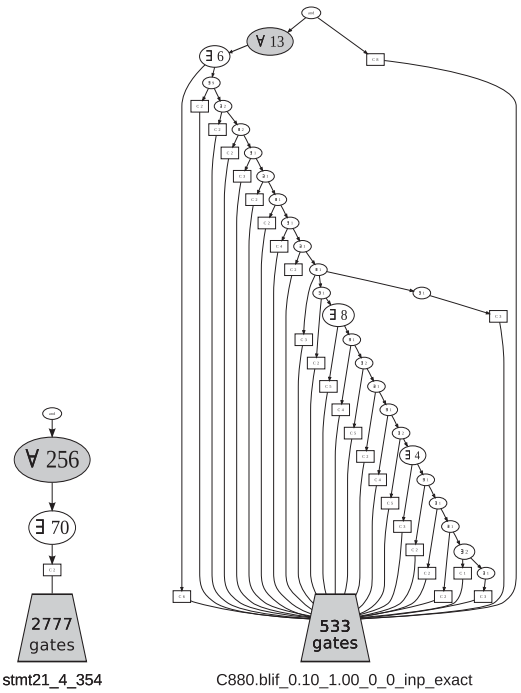

Figure 1: Quantifier Trees

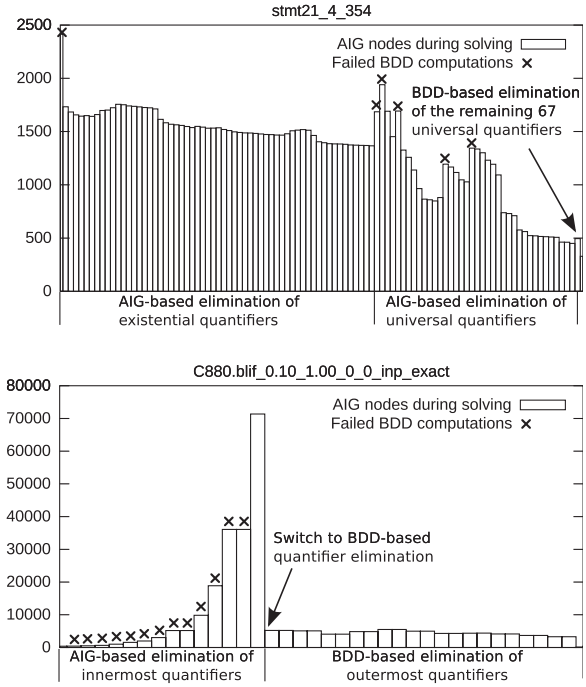

Figure 2: Development of AIG node counts

tifications, the solver tries to compute BDD representations which again fails due to resource limits (Crosses in Fig. 2 mark failed BDD computations). At the point where only 67 universal quantifiers are left, BDD computation finally succeeds and the remaining quantifiers can be eliminated by BDD operations.

On the *C880.blif_0.10_1.00_0_0_inp_exact* instance, which is *hard* according to *QBF Evaluation*,[4] the solver's behavior is completely different. Here, the preprocessor reduces the number of variables from 1022 to 619 and the number of clauses from 6007 to 4201. Then 533 functional definitions (527 AND-gates and 6 XOR-gates) are detected and extracted from the formula. The resulting QBF structure after quantifier tree computation is shown in Fig. 1 (right hand side). Again, the solver starts to eliminate quantifiers bottom-up, first trying to compute a BDD representation, which fails, therefore AIG-based quantifier elimination is used. Unfortunately, optimization techniques are not able to compress the AIG representations such that the number of AIG nodes quickly grows. Due to the growth, *AIGSolve* tries to compute BDDs for the intermediate AIGs, which fails 14 times, forcing the solver to continue AIG-based elimination for the innermost 14 quantifiers. Finally, after eliminating 14 quantifiers, the computation of a BDD succeeds for an AIG containing 71287 nodes. The remaining quantifiers are all eliminated by BDD operations.

These two examples with completely different characteristics illustrate that a tight interaction between AIG and BDD based methods is indeed crucial for the success of the method.

5. EXPERIMENTAL RESULTS

5.1 Setup

For all experiments we used the complete benchmark set of the *QBF Evaluation* 2008 [22] consisting of 3328 benchmarks from various application domains. All benchmarks were run on a 16 core AMD Opteron with 2.3 GHz and 64 GB of memory[5]. We used a time limit of 600 CPU seconds to be consistent with the result of the *QBF Evaluation*, failed instances (violation of time or memory limits) are considered to contribute 600 CPU seconds to the overall run time.

We extended the AIG-based solver *AIGSolve*1.0 presented in [23] by the methods described in this paper and refer to the resulting solver as *AIGSolve*2.0.

5.2 Effect on preprocessing

In order to evaluate the effectiveness of the proposed SAT-based constant detection technique, we first applied three variants of *AIGSolve*2.0's preprocessing component on the complete *QBF Evaluation* 2008 benchmark set: one which does not perform SAT-based constant detection at all ("No Constant Detection"), one which additionally includes SAT-based constant detection but with the original decision heuristics ("Constant Detection (original decision heuristics)"), and one which includes constant detection along with the proposed modification of the decision heuristics ("Constant Detection (modified decision heuristics)").

Table 1 shows the results grouped by benchmark classes: for each benchmark class ("class", with "count" instances), the table lists the number of instances completely solved by each preprocessor variant ("solved"). Columns "preproc. time" and "const. det. time" give the average total preprocessing times and the average times used for SAT-based constant detection in CPU seconds. To measure the effect of learning from satisfying assignments during constant detection, the table also lists the average percentages of avoided SAT checks ("avoided checks"). The global effect of constant detection is shown in the remaining two columns: column "constants" lists the average ratio of detected constants and initial variables of the instance, column "eliminated variables" denotes the average percentages of formula variables eliminated by the whole preprocessing procedure (including unit propagation, detection of equivalent literals etc.).

The preprocessor variant without constant detection is able to preprocess the benchmarks in an average of 3.0 CPU seconds, already solving a total of 493 instances. On average 25.4% of the initial variables are eliminated by the preprocessor.

Including the SAT-based constant detection method into the pre-

[4]This means that no solver taking part in the evaluation was able to solve these instances within the timeout of 600 CPU seconds.

[5]The preprocessors and solvers were run as 32 bit processes, so the memory was effectively limited to 4 GB.

		No Constant Detection			Constant Detection (original decision heuristics)						Constant Detection (modified decision heuristics)					
class	count	solved	preproc. time	elim. variables	solved	preproc. time	const. det. time	avoided checks	constants	elim. variables	solved	preproc. time	const. det. time	avoided checks	constants	elim. variables
abduction	303	1	6.0	2.8%	1	9.0	3.1	8.5%	15.9%	21.4%	1	6.1	0.1	87.2%	15.9%	21.4%
adder	32	0	0.0	11.9%	0	5.0	4.9	46.5%	0.0%	11.9%	0	0.4	0.3	98.2%	0.0%	11.9%
bbox	478	110	0.3	53.1%	110	11.4	11.1	32.6%	0.4%	54.2%	110	5.5	5.1	96.3%	0.6%	54.3%
blocks	13	8	0.0	74.8%	8	0.1	0.1	48.9%	9.4%	82.0%	8	0.1	0.0	77.8%	9.4%	82.0%
bmc	132	109	130.9	82.6%	108	144.3	12.4	56.2%	1.5%	81.8%	110	130.5	5.8	95.5%	1.6%	83.3%
circuits	63	8	195.8	39.7%	8	198.4	9.2	62.6%	0.0%	39.7%	8	190.1	2.3	98.6%	0.0%	39.7%
counter	24	8	0.1	34.1%	8	4.2	4.1	60.1%	0.0%	34.1%	8	0.2	0.2	94.9%	0.0%	34.1%
debug	38	0	456.5	1.1%	0	474.7	7.9	67.8%	0.0%	1.0%	0	471.1	6.5	89.0%	0.0%	1.1%
ev-pr	38	0	17.6	11.1%	0	34.8	13.6	10.4%	0.0%	11.7%	0	17.9	0.8	95.1%	0.0%	11.7%
fpga	8	2	5.1	25.6%	2	7.0	0.0	34.8%	0.0%	25.6%	2	6.2	0.0	88.5%	0.0%	25.6%
jmc-quant+sqr	20	0	0.0	6.3%	0	2.2	2.2	47.5%	0.1%	6.5%	0	0.2	0.1	95.6%	0.1%	6.5%
k_*	378	18	0.2	5.8%	44	3.9	3.6	76.9%	1.1%	19.5%	44	2.7	2.4	88.1%	1.1%	20.0%
planning	24	6	208.1	25.2%	5	256.1	5.0	58.7%	2.4%	21.8%	5	226.3	3.1	92.7%	2.4%	21.9%
qshifter	6	0	120.8	0.0%	0	121.3	0.0	47.5%	0.0%	0.0%	0	119.3	0.0	95.0%	0.0%	0.0%
scholl-becker	64	32	0.5	61.7%	32	3.4	3.3	50.8%	4.2%	62.4%	32	1.7	1.6	90.0%	4.2%	62.4%
sorting	84	10	0.0	21.1%	10	4.6	4.6	39.3%	0.0%	21.1%	10	0.3	0.2	89.3%	0.0%	21.1%
s_*	171	0	0.6	13.7%	0	11.9	11.1	91.5%	1.1%	15.2%	0	7.0	6.3	97.2%	1.3%	15.5%
stmt	713	6	0.1	2.5%	6	6.0	6.0	48.2%	0.5%	3.2%	6	0.7	0.6	96.5%	0.6%	3.3%
szymanski	12	0	0.7	0.3%	0	17.3	16.5	36.9%	0.1%	0.5%	0	12.2	11.3	95.3%	0.1%	0.5%
tipdiameter	203	36	0.2	28.5%	36	6.5	6.3	33.3%	9.3%	39.0%	36	2.3	2.1	88.2%	12.6%	44.6%
tipfixpoint	446	69	0.3	19.7%	69	13.6	13.2	25.3%	3.7%	26.5%	69	4.3	4.1	94.3%	7.2%	32.2%
other	78	70	5.2	90.3%	70	8.3	2.1	57.2%	2.2%	90.3%	70	5.3	0.5	76.6%	2.3%	90.3%
total	3328	493	3.0	25.4%	517	10.3	7.6	44.3%	3.1%	30.7%	519	5.4	2.6	92.9%	3.8%	31.9%

Table 1: Effects of Constant Detection on Preprocessing

processor (but not using the modified decision heuristics) increases the average preprocessing time to 10.3 CPU seconds, on average 7.6 CPU seconds are used for constant detection. The constant detection component of the preprocessor is able to prove that 3.1% of the initial variables are constants, on average eliminating 30.7% of all variables from the instances. Note that successful constant detection may trigger further simplification steps in the preprocessor such that the percentage of eliminated variables may be higher than the sum of the percentages of eliminated variables without constant detection and the percentage of detected constants. However, the percentage of avoided SAT-checks is only 44.3%, which indicates that learning from satisfying assignments during constant detection is not very efficient when relying on the default decision heuristics of the SAT-solver.

Using the modified decision heuristics considerably increases the quality of satisfying assignments returned by the SAT-solver: the percentage of avoided SAT checks raises to 92.9% while the time used for constant detection decreases to 2.6 CPU seconds. Due to faster constant detection, the time limit for constant detection is less often violated, resulting in more variables to be detected as constants (3.8%) and more variables to be removed from the instances (31.9%). Moreover, 519 instances are solved just by applying preprocessing.

On some benchmark classes constant detection is especially successful: for *abduction* 15.9% of the variables are proven to be constant, the *tipdiameter* benchmarks contain 12.6% constant variables, in *blocks* 9.4% constants are detected. Conversely, there are some benchmark classes, where no constants could be detected, including *adder*, *circuits*, and *sorting*.

Altogether, constant detection in combination with a clever modification of the decision heuristics notably improves the preprocessor in terms of eliminating variables from the preprocessed QBF instance while leading to only a small increase in runtime.

5.3 Effect on solving and comparison with state-of-the-art solvers

In a second experiment, we wanted to find out if the additional effort for detecting constant literals and our modifications to the interaction between AIG and BDD based solving techniques have a positive effect on the performance of the actual solver component of *AIGSolve*2.0. Again we used multiple preprocessor variants: one with ("*AIGSolve*2.0 (CD)") and without constant detection ("*AIGSolve*2.0 (no CD)"). For the variant with constant detection, we also applied modification of the decision heuristics. For comparison, we ran the latest versions of the two search-based QBF-solvers *yQuaffle* [31] and *QuBE* [13], as well as the two elimination based solvers *Quantor* [4] and *sKizzo* [3] on the benchmark set. This selection of solvers includes the winners of the previous two QBF evaluations (*sKizzo* and *Quantor*), as well as the most powerful search-based solver according to *QBF Evaluation* 2008 (*QuBE*). Furthermore, to capture the effect of the other modifications to *AIGSolve* that we described in section 4, we also display results for *AIGSolve*1.0. The results are shown in table 2: for each solver and each benchmark class the column "solved" lists the number of solved instances, the column "time" displays the total amount of used time in CPU hours.

*AIGSolve*1.0 solves a total of 1976 instances, which lies between the numbers of instances solved by *sKizzo* (1692) and *QuBE* (2205), but which is considerably larger than the numbers achieved by the solvers *Quantor* (973) and *yQuaffle* (923).

Applying the modifications to the quantification algorithm and to the preprocessor (expansion of universal variables) (leading to *AIGSolve*2.0 (without constant detection)) improves performance in 19 benchmark classes, such that it is able to solve 2195 instances in total which is slightly below the *QuBE*'s success rate. A closer look at the the individual solver runs reveals that the large increase of solved instances in the "bmc" class is due to the expansion of universal variables, whereas the increase in other classes results from the modified quantifier elimination algorithm.

Including the SAT-based constant detection method, lets *AIGSolve*2.0 improve its success rate in 9 of the 22 benchmark classes. Altogether it solves 2370 benchmarks and thus clearly outperforms all other solvers. The largest gain is achieved in the class *abduction* where constant detection was able to find the largest number of constants. But also on other classes (especially *tipfixpoint*, *stmt* and *k_**) the solver's performance increases notably.

Comparing the best version of *AIGSolve*2.0 with the four other solvers, one can furthermore observe that *AIGSolve*2.0 has a large number of uniquely solved instances: it uniquely succeeds on 261

		yQuaffle		Quantor		sKizzo		QuBE		AIGSolve1.0		AIGSolve2.0 (no CD)		AIGSolve2.0 (CD)	
class	count	solved	time	solved	time	solved	time	solved	time	solved	time	solved	time	solved	time
abduction	303	260	7.4	80	38.9	171	23.1	**286**	3.2	78	38.6	79	38.4	196	20.0
adder	32	4	4.7	8	4.0	10	3.7	5	4.5	19	2.8	**24**	1.7	**24**	1.7
bbox	478	142	56.0	130	58.0	171	51.5	**388**	16.8	196	48.2	205	45.9	212	44.7
blocks	13	11	0.4	**13**	0.0	8	0.8	7	1.0	6	1.2	8	0.8	8	0.8
bmc	132	70	11.2	**115**	3.4	86	7.9	57	13.1	30	17.2	109	4.8	110	4.7
circuits	63	4	9.8	**8**	9.3	6	9.5	4	9.8	4	9.9	**8**	9.3	**8**	9.2
counter	24	9	2.5	12	2.0	12	2.0	9	2.5	13	2.2	**16**	1.5	**16**	1.5
debug	38	0	6.3	**33**	2.1	0	6.3	0	6.3	0	6.3	0	6.3	0	6.3
ev-pr	38	7	5.2	1	6.2	10	4.7	**17**	3.8	0	6.3	1	6.3	2	6.2
fpga	8	7	0.2	**8**	0.0	7	0.2	6	0.4	6	0.6	5	0.5	5	0.5
jmc-quant+sqr	20	0	3.3	0	3.3	0	3.3	5	2.7	11	1.8	**16**	0.7	**16**	0.7
k_*	378	133	42.3	259	20.3	281	16.8	209	29.2	335	9.2	338	7.9	**350**	6.0
planning	24	8	2.8	**15**	1.8	4	3.3	6	3.1	1	3.8	9	2.7	10	2.8
qshifter	6	1	0.8	**6**	0.0	**6**	0.0	5	0.2	**6**	0.0	5	0.2	5	0.2
scholl-becker	64	36	5.1	38	4.8	32	5.3	37	4.8	46	3.2	**50**	2.2	**50**	2.4
sorting	84	25	10.2	**39**	7.6	12	12.0	33	8.8	10	12.8	11	12.2	11	12.2
s_*	171	1	28.3	17	26.1	13	26.4	62	19.6	33	24.2	**66**	20.3	**66**	20.3
stmt	713	9	117.3	21	115.5	633	14.8	610	17.4	653	17.0	665	12.2	**682**	8.8
szymanski	12	0	2.0	3	1.5	5	1.2	**12**	0.0	3	1.5	5	1.2	5	1.2
tipdiameter	203	68	22.8	77	21.3	96	18.3	157	8.4	184	4.1	185	3.4	**190**	2.7
tipfixpoint	446	66	63.4	19	71.2	54	66.0	222	39.9	270	32.3	314	24.7	**328**	21.8
other	78	62	2.9	71	1.2	75	0.5	68	1.8	72	1.2	**76**	0.5	**76**	0.5
total	3328	923	405.1	973	398.7	1692	277.8	2205	197.4	1976	244.4	2195	203.3	**2370**	175.1

Table 2: Comparison with other Solvers

benchmarks followed by *QuBE* (238), *Quantor* (72), *sKizzo* (3), and *yQuaffle* (3).

247 of the instances solved by *AIGSolve*2.0 have also been rated "hard" at the *QBF Evaluation* 2008, which indicates that no solver taking part in the evaluation was able to solve these instances within 600 CPU seconds.

6. CONCLUSIONS

In this paper, we presented a novel preprocessing technique for QBF deploying incremental SAT-checks to prove that variables are constant in all satisfying assignments and thus can be eliminated. As shown in the experiments, the key component of this method is the exploitation of satisfying assignments computed by the SAT-solver, which is especially effective when the solver's decision heuristics are modified to produce "helpful" assignments. Furthermore, we described an original hybrid quantifier elimination scheme for AIGs, which combines AIG and BDD-based quantification to profit from both representations. Experimental results on the instances from the *QBF Evaluation* 2008 show that the integration of the presented methods into the QBF solver *AIGSolve* considerably increases the solver's performance and lets it significantly outperform other state-of-the-art solvers.

For the future we plan to extend the hybrid quantifier elimination algorithm by integrating and adapting other promising techniques such as methods based on functional composition [14].

7. REFERENCES

[1] P. A. Abdulla, P. Bjesse, and N. Eén. Symbolic Reachability Analysis Based on SAT-Solvers. In *Proc. of TACAS 2000*.

[2] A. Ayari and D. A. Basin. QUBOS: Deciding Quantified Boolean Logic Using Propositional Satisfiability Solvers. In *Proc. of FMCAD 2002*.

[3] M. Benedetti. sKizzo: A Suite to Evaluate and Certify QBFs. In *Proc. of CADE 2005*.

[4] A. Biere. Resolve and Expand. In *Proc. of SAT 2004, Selected Papers*.

[5] R. Bryant. Graph - Based Algorithms for Boolean Function Manipulation. *IEEE Trans. on Comp.*, 35(8):677–691, 1986.

[6] M. Cadoli, M. Schaerf, A. Giovanardi, and M. Giovanardi. An algorithm to evaluate quantified Boolean formulae. In *Journal of Automated Reasoning*, pages 262–267, 1998.

[7] M. Davis, G. Logemann, and D. W. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962.

[8] M. Davis and H. Putnam. A Computing Procedure for Quantification Theory. *Journal of the ACM*, 7(3):201–215, 1960.

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

[10] N. Eén and A. Biere. Effective Preprocessing in SAT Through Variable and Clause Elimination. In *Proc. of SAT 2005*.

[11] N. Eén and N. Sörensson. An Extensible SAT-solver. In *Proc. of SAT 2003*.

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

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

[14] J.-H. R. Jiang. Quantifier Elimination via Functional Composition. In *Proc. of CAV 2009*.

[15] T. Jussila and A. Biere. Compressing BMC Encodings with QBF. *Electr. Notes Theor. Comput. Sci.*, 174(3):45–56, 2007.

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

[17] W. Kunz and D. K. Pradhan. Recursive learning: a new implication technique for efficient solutions to CAD problems-test, verification, and optimization. *IEEE TCAD*, 13(9):1143–1158, 1994.

[18] C. M. Li and A. Anbulagan. Heuristics based on unit propagation for satisfiability problems. In *Proc. of IJCAI 1997*, San Francisco, CA, USA.

[19] A. Mishchenko, S. Chatterjee, and R. Brayton. DAG-aware AIG rewriting a fresh look at combinational logic synthesis. In *Proc. of DAC 2006*.

[20] A. Mishchenko, S. Chatterjee, R. Jiang, and R. K. Brayton. FRAIGs: A unifying representation for logic synthesis and verification. Technical report, EECS Dept., UC Berkeley, 03 2005.

[21] G. Pan and M. Y. Vardi. Symbolic Decision Procedures for QBF. In *Proc. of CP 2004*.

[22] C. Peschiera, A. Tacchella, and A. Tacchella. QBF Evaluation 2008. `http://www.qbfeval.org/2008`.

[23] F. Pigorsch and C. Scholl. Exploiting structure in an AIG based QBF solver. In *Proc. of DATE 2009*.

[24] F. Pigorsch, C. Scholl, and S. Disch. Advanced Unbounded Model Checking Based on AIGs, BDD Sweeping, And Quantifier Scheduling. In *Proc. of FMCAD 2006*.

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

[26] H. Samulowitz and F. Bacchus. Binary Clause Reasoning in QBF. In *Proc. of SAT 2006*.

[27] H. Samulowitz, J. Davies, and F. Bacchus. Preprocessing QBF. In *Proc. of CP 2006*.

[28] C. Scholl and B. Becker. Checking Equivalence for Partial Implementations. In *Proc. of DAC 2001*.

[29] P. Simons. Extending the Stable Model Semantics with More Expressive Rules. In *Proc. of LPNMR 1999*.

[30] J. Whittemore, J. Kim, and K. A. Sakallah. SATIRE: A New Incremental Satisfiability Engine. In *Proc. of DAC 2001*.

[31] L. Zhang and S. Malik. Conflict driven learning in a quantified Boolean Satisfiability solver. In *Proc. of ICCAD 2002*.

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