Preprint from of the 4th International Workshop on Quantified Boolean Formulas (QBF 2016), co-located with SAT 2016, Bordeaux, France, July 4, 2016.

The QBF Solver AIGSolve

Christoph Scholl and Florian Pigorsch

Albert-Ludwigs-Universität Freiburg im Breisgau, Germany
{scholl,pigorsch}@informatik.uni-freiburg.de

Abstract. AIGSolve[1] is a rewriting-based solver based on And-Inverter Graphs (AIGs). In this approach, quantifiers are eliminated, starting with the inner-most quantifiers. Intermediate results are represented symbolically using AIGs [22, 23]. The basic method consists of cofactor-based quantifier elimination which is combined with a multitude of optimization approaches including a SAT- and BDD-based compaction of the representations, methods for preprocessing QBF-formulas based on incremental SAT, heuristics that exchange quantifiers of the same level, heuristics that use BDD-based quantifier elimination as an alternative, as well as structure extraction and structure exploitation for the processed instances.

1 Introduction

Quantified Boolean Formulas (QBF) are a powerful generalization of propositional formulas (or SAT formulas). In contrast to SAT formulas, QBF allows existentially as well as universally quantified variables, which potentially allows for exponentially more compact representations of many problems compared to SAT formulas, but comes at the price of raising the decision complexity from NP-complete to PSPACE-complete.

Many real world problems from various application domains, such as formal verification and artificial intelligence, can be compactly formulated as QBF, including the verification of incomplete circuit designs [29, 12], conditional planning [26], and nonmonotonic reasoning problems [9].

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 [11, 30, 17, 18] which apply an extension of the DPLL algorithm known from SAT [7], as well as solvers based on eliminating variables by different methods. Concerning variable elimination we can differentiate between a *eager* elimination by methods like symbolic skolemization [2], resolution and expansion [1, 3], and symbolic quantifier elimination using AIGs [22, 23] on the one hand and *lazy*, CEGAR-based expansion as in [14, 13]. AIGSolve is based on eager variable elimination using AIGs (And-Inverter Graphs) as the internal representation format.

The common format for QBF instances is the prenex conjunctive normal form, which consists of a linear quantifier prefix and a propositional part in CNF

[1] AIGSolve is available for download at [21].

format. General QBFs from the application domain are typically transformed to the prenex format in a two staged process: first, quantifiers are pushed outwards the formula, leaving an arbitrarily shaped propositional part. In a second step, this propositional part is encoded as a CNF formula. Especially for our approach it is beneficial to extract structural information from the CNF part of a prenex QBF instance before the solving process. Irrespective of the question whether the CNF part originally resulted from a structural circuit description or not – we try to detect and extract clauses from the CNF part of a prenex QBF instance which establish functional definitions of variables. Then we use these definitions to generate a non-CNF QBF formula, and directly represent it by a symbolic data-structure based on And-Inverter Graphs (AIGs) [16]. Finally, the actual QBF solving process is performed by eliminating quantifiers using specialized AIG operations and/or BDD operations.

2 And-Inverter Graphs

In our approach we are using And-Inverter Graphs (AIGs) [16], or more precisely Functionally Reduced AIGs (FRAIGs) [20, 24], as a compact symbolic representation for propositional formulas. AIGs are boolean circuits composed solely of two-input AND gates and inverters. In contrast to BDDs [4], they are not a canonical representation for boolean functions – for each boolean function there exist many structurally different AIGs. Actually an AIG may contain functionally redundant nodes, i. e., nodes which are roots of structurally different subgraphs representing the same functions. A restriction of general AIGs are FRAIGs, which are 'semi-canonical' by prohibiting nodes which represent the same (or inverse) boolean functions. This property is called 'functional reduction property'. To achieve this property several techniques like structural hashing, simulation and SAT solving are used during FRAIG construction.

3 Preprocessing

As many other solvers *AIGSolve* starts with a preprocessing phase. In the main loop of the preprocessing phase, unit propagation [6], subsumption checking [8], for-all reduction [15, 28], and equivalence reduction [27] are applied until closure.

Then, a SAT-based *constant detection* [23] follows which is embedded into the main preprocessing loop as depicted in Alg. 1. For a QBF $\psi = \mathcal{Q}_1 x_1 \ldots \mathcal{Q}_n x_n.\ \phi(x_1, \ldots, x_n)$ and a literal ℓ_i in the QBF, constant detection checks whether $\phi(x_1, \ldots, x_n) \rightarrow \ell_i$. If this is the case, then $\psi' = \mathcal{Q}_1 x_1 \ldots \mathcal{Q}_n x_n.\phi(x_1, \ldots, x_n) \wedge \ell_i$ is equivalent to ψ. Constant detection for all possible literals can be reduced to a series of incremental SAT checks and can be improved by using satisfying assignments for satisfied instances during this process [23]. The time the solver is allowed to spend in constant detection is strictly limited. If the time for constant detection is exceeded, constant detection is aborted and the set of constants discovered until this point is returned.

Moreover, *AIGSolve*'s preprocessing routine performs a complete expansion of the universal variables [1, 3] 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 on instances with few universal variables, the resulting formulas are of moderate size and can often be solved efficiently by SAT-solvers. If the QBF has not been fully expanded, trivial SAT checks (omitting the quantifier prefix) are finally performed to check whether the matrix of the formula is unsatisfiable or a tautology [5].

Algorithm 1 Preprocessor for QBF Q.

```
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);
  return SatSolve(Q);
else
  TrivialSatisfiability(Q);
end if
return Q;
```

4 AIG-Based Solving with BDD Support

4.1 Structure Extraction and Moving Quantifiers

After preprocessing, *AIGSolve* scans the remaining QBF formula for clausal gate definitions as it is done for example in SAT preprocessing [8] or by other QBF preprocessors and solvers [3, 10] 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* [2].

Example 1. As an example, consider the following QBF:

$$\forall a \exists b \exists c \forall d \exists e \exists f . (\overline{a} \vee b) \wedge (a \vee e) \wedge (\overline{c} \vee a) \wedge (\overline{c} \vee b) \\ \wedge (c \vee \overline{a} \vee \overline{b}) \wedge (c \vee d \vee f) \wedge (e \vee b \vee c) \wedge (c \vee \overline{f})$$

We detect a definition for c: $c \leftrightarrow a \wedge b$ using the definition clauses $(\overline{c} \vee a)$, $(\overline{c} \vee b)$, and $(c \vee \overline{a} \vee \overline{b})$. We remove the definition clauses and structurally replace c by $a \wedge b$, leading to a representation as depicted in Fig. 1. The connections from c to $a \wedge b$ denote the association between the variable and its definition.

Then, moving the quantifiers for early quantification results in the (generalized) quantifier tree with clauses and functional definitions as depicted in Fig. 2. Again, the variable c is linked to its definition $a \wedge b$.

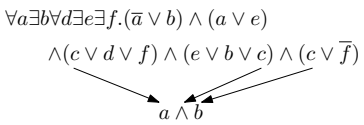

Fig. 1. After structure extraction.

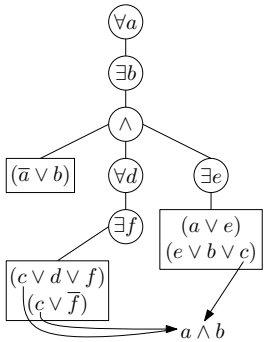

Fig. 2. After quantifier tree construction.

4.2 Quantifier Elimination

AIGSolve then transforms the generalized quantifier tree (as in Ex. 1) into an AIG representation and eliminates all quantifiers in a bottom-up fashion. The basic operation consists in performing cofactoring on AIG cones (existential quantification wrt. x_i leads to a disjunction of positive and negative cofactors wrt. x_i, universal quantification to a corresponding conjunction), compressing the individual results of quantifier elimination by BDD-sweeping [24], functional reduction [20] and DAG-aware rewriting [19] of the AIG structure. Moreover, in case of several quantifiers in series in a quantifier block, the quantifications can be exchanged and are performed using the AIG-size based quantifier scheduling heuristics from [24].

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. If it is possible to build a small BDD for some AIG cone, we allow an extended exploitation of good BDD representations as follows:

Given an AIG f and a variable x which is to be eliminated, 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.

5 Experiments

Experimental evaluations of *AIGSolve* can be found in the original papers [22, 23]. A more extensive and recent evaluation can be found at [25]. Instead of repeating those results, we demonstrate the interaction between AIG- and BDD-based quantifier elimination methods by looking into the solver's behavior on two benchmarks from *QBF Evaluation* 2008, see also [23].

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. 3. 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.

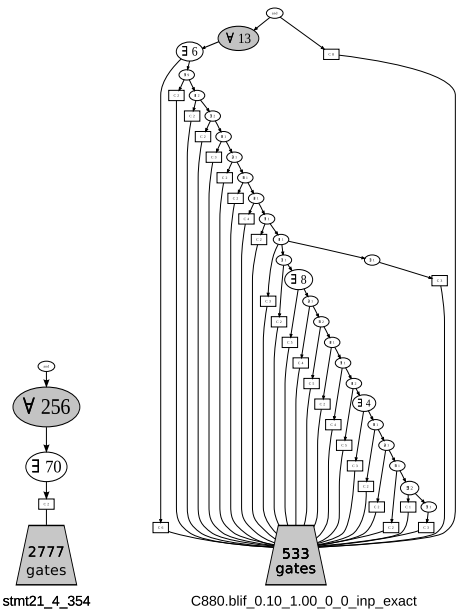

Fig. 3. Quantifier Trees

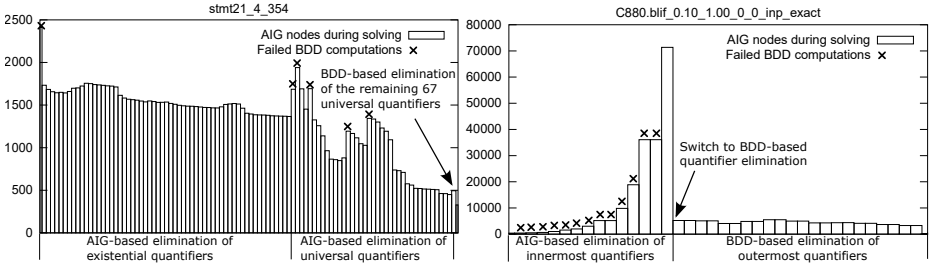

Fig. 4. Development of AIG node counts

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. 4 (left

hand side). 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 quantifications, the solver tries to compute BDD representations which again fails due to resource limits (Crosses in Fig. 4 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* 2008[2], 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. 3 (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.

References

1. Ayari, A., Basin, D.A.: QUBOS: Deciding Quantified Boolean Logic Using Propositional Satisfiability Solvers. In: Proc. of FMCAD (2002)
2. Benedetti, M.: sKizzo: A Suite to Evaluate and Certify QBFs. In: Proc. of CADE (2005)
3. Biere, A.: Resolve and Expand. In: Proc. of SAT (2004)
4. Bryant, R.: Graph - Based Algorithms for Boolean Function Manipulation. IEEE Trans. on Comp. 35(8), 677–691 (1986)

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

5. Cadoli, M., Schaerf, M., Giovanardi, A., Giovanardi, M.: An algorithm to evaluate quantified Boolean formulae. In: Journal of Automated Reasoning. pp. 262–267 (1998)
6. Davis, M., Putnam, H.: A Computing Procedure for Quantification Theory. Journal of the ACM 7(3), 201–215 (1960)
7. Davis, M., Logemann, G., Loveland, D.W.: A machine program for theorem-proving. Commun. ACM 5(7), 394–397 (1962)
8. Eén, N., Biere, A.: Effective Preprocessing in SAT Through Variable and Clause Elimination. In: Proc. of SAT (2005)
9. Egly, U., Eiter, T., Tompits, H., Woltran, S.: Solving Advanced Reasoning Tasks Using Quantified Boolean Formulas. In: Proc. of AAAI/IAAI (2000)
10. Giunchiglia, E., Marin, P., Narizzano, M.: sQueezBF: An Effective Preprocessor for QBF. In: Proc. of QiCP (2008)
11. Giunchiglia, E., Narizzano, M., Tacchella, A.: QUBE: A System for Deciding Quantified Boolean Formulas Satisfiability. In: Proc. of IJCAR (2001)
12. Herbstritt, M., Becker, B.: On Combining 01X-Logic and QBF. In: Proc. of EUROCAST (2007)
13. Janota, M., Klieber, W., Marques-Silva, J., Clarke, E.M.: Solving QBF with counterexample guided refinement. In: Proc. of SAT. pp. 114–128 (2012)
14. Janota, M., Silva, J.P.M.: Abstraction-based algorithm for 2qbf. In: Proc. of SAT. pp. 230–244 (2011)
15. Kleine Büning, H., Karpinski, M., Flögel, A.: Resolution for quantified boolean formulas. Inf. Comput. 117(1), 12–18 (1995)
16. Kuehlmann, A., Ganai, M.K., Paruthi, V.: Circuit-based Boolean Reasoning. In: Proc. of DAC (2001)
17. Letz, R.: Lemma and model caching in decision procedures for quantified boolean formulas. In: Proc. of TABLEAUX 2002. pp. 160–175 (2002)
18. Lonsing, F., Biere, A.: DepQBF: A dependency-aware QBF solver. Journal on Satisfiability, Boolean Modelling and Computation 7(2-3), 71–76 (2010)
19. Mishchenko, A., Chatterjee, S., Brayton, R.: DAG-aware AIG rewriting a fresh look at combinational logic synthesis. In: Proc. of DAC (2006)
20. Mishchenko, A., Chatterjee, S., Jiang, R., Brayton, R.K.: FRAIGs: A unifying representation for logic synthesis and verification. Tech. rep., EECS Dept., UC Berkeley (03 2005)
21. Pigorsch, F., Scholl, C.: AIGSolve, `http://abs.informatik.uni-freiburg.de/src/tools.php`
22. Pigorsch, F., Scholl, C.: Exploiting structure in an AIG based QBF solver. In: Proc. of DATE. pp. 1596–1601 (2009)
23. Pigorsch, F., Scholl, C.: An AIG-based QBF-solver using SAT for preprocessing. In: Proc. of DAC. pp. 170–175 (2010)
24. Pigorsch, F., Scholl, C., Disch, S.: Advanced Unbounded Model Checking Based on AIGs, BDD Sweeping, And Quantifier Scheduling. In: Proc. of FMCAD (2006)
25. Pulina, L.: QBF Evaluation 2016, `http://www.qbflib.org/index_eval.php`
26. Rintanen, J.: Constructing Conditional Plans by a Theorem-Prover. J. Artif. Intell. Res. (JAIR) 10, 323–352 (1999)
27. Samulowitz, H., Bacchus, F.: Binary Clause Reasoning in QBF. In: Proc. of SAT (2006)
28. Samulowitz, H., Davies, J., Bacchus, F.: Preprocessing QBF. In: Proc. of CP (2006)
29. Scholl, C., Becker, B.: Checking Equivalence for Partial Implementations. In: Proc. of DAC (2001)
30. Zhang, L., Malik, S.: Towards Symmetric Treatment of Conflicts And Satisfaction in Quantified Boolean Satisfiability Solver. In: Proc. of CP (2002)