Exploiting Structure in an AIG Based QBF Solver

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

***Abstract*—In this paper we present a procedure for solving quantified boolean formulas (QBF), which uses And-Inverter Graphs (AIGs) as the core data-structure. We make extensive use of structural information extracted from the input formula such as functional definitions of variables and non-linear quantifier structures. We show how this information can directly be exploited by the symbolic, AIG based representation. We implemented a prototype QBF solver based on our ideas and performed a number of experiments proving the effectiveness of our approach, and moreover, showing that our method is able to solve QBF instances on which state-of-the-art QBF solvers known from literature fail.**

I. Introduction

Quantified Boolean Formulas (QBF) are a powerful generalization of satisfiability formulas (SAT). In contrast to SAT, QBF allows existentially as well as universally quantified variables, which allows for exponentially more compact representations of many problems compared to SAT, 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 [1], [2], conditional planning [3], and nonmonotonic reasoning problems [4].

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

However, previous work showed that QBF solving often performs much better for non-prenex formats with quantifier trees instead of linear quantifier prefixes at the beginning of the formula [5], [6].

In this paper, we will show that it is also beneficial to extract structural information from the CNF part of a prenex QBF instance before the solving process. As we will see later on in the paper, this is in particular true for our AIG based solver. 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) [7]. Finally, the actual QBF solving process is performed by eliminating quantifiers using specialized AIG operations.

The paper is organized as follows: Sect. II gives a short introduction to QBF and the symbolic representation used in our solver. Sect. III describes the components of our approach in detail: i. e. the preprocessing in section III-A, the structure extraction method in Sect. III-B, early quantification in Sect. III-C, and finally the symbolic quantifier elimination part in Sect. III-D. In Sect. IV we evaluate the effectiveness of our approach, and conclude the paper in Sect. V.

Related Work

In the SAT domain, structure extraction and exploitation has been successfully studied by several authors, e.g., [8], [9]. Since they are restricted to SAT problems, both approaches do not consider quantification levels and quantifier types of the involved variables which is needed for a correct extraction in the QBF context. In [8] the authors propose a CNF preprocessor that uses functional definitions to eliminate variables from the CNF, but they still stick with a (potentially larger) CNF representation after variable elimination. The authors of [9] extract functional definitions, and generate a non-CNF representation equivalent to the input CNF. During SAT solving the non-CNF parts are used in order to reject satisfying assignments for the CNF part which are not consistent with functional definitions.

To the best of our knowledge extraction and exploitation of structural knowledge in the QBF solving domain has only been considered by approaches working on CNF representations of the QBF formula:

In [6] the authors present a CNF-based QBF solver discovering non-linear quantifier structures from prenex QBF instances and utilizing these structures in their solving algorithm. Additionally functional definitions are used to support the elimination of variables. The authors of [10] similarly exploit functional definitions in their QBF preprocessor. Both approaches rely on CNF representations, and thus suffer from the potential increase of the CNF during the elimination steps.

The potential of non-CNF representations for QBF is demonstrated in several works: In [11] the authors argue that CNF representations are inherently limiting for the QBF solution process and propose a search-based solver working on a combination of CNF and DNF. The authors of [12] use tree-shaped negation normal forms for QBF solving which allow a more compact representation of the results of variable expansion than CNF. In [13], [5] a procedure based on quantifier

tree reconstruction and symbolic skolemization is presented. Intermediate skolem functions are represented using BDDs. However, in all three approaches the possibility to benefit from functional definitions is not considered.

In [14] a search-based method is presented to operate directly on *non-prenex* CNF instances, but, again, functional definitions are not exploited.

II. Preliminaries

A. Quantified Boolean Formulas

In this work we deal with several types of Quantified Boolean Formulas: (1) general QBF, (2) prenex QBF, and (3) QCNF.

The least restricted type of QBF is the *general QBF*, which is an arbitrary propositional formula consisting of Boolean variables and Boolean operators, combined with existential and universal quantifiers at arbitrary positions.

A specialization of the general QBF type is the *prenex QBF*, which is of the form $Q.\phi$, where ϕ is a propositional formula over a set of variables V and $Q = \mathcal{Q}_1 V_1 \mathcal{Q}_2 V_2 \dots \mathcal{Q}_n V_n$ is a sequence of alternating quantifiers, such that $\mathcal{Q}_i \in \{\forall, \exists\}$, $\mathcal{Q}_i \neq \mathcal{Q}_{i+1}$, $V_i \neq \emptyset$, $V = \bigcup_{i=1}^{n} V_i$, and $V_i \cap V_j = \emptyset$. Q is called *prefix* and ϕ is called the *matrix* of the QBF. A variable $v \in V_i$ is *existential* (*universal*) if the $\mathcal{Q}_i = \exists$ ($\mathcal{Q}_i = \forall$). The *quantification level* of a variable is $\lambda(v) = i \Leftrightarrow v \in V_i$.

If the matrix of a prenex QBF is in conjunctive normal form (*CNF*), we are talking of a *QCNF* formula. Most QBF solvers accept input in form of QCNF formulas and also use QCNF like data structures for the internal representations.

B. And-Inverter Graphs

In our approach we are using And-Inverter Graphs (AIGs) [7], or more precisely Functionally Reduced AIGs (FRAIGs) [15], [16], as a compact symbolic representation for satisfying assignments of QBF formulas. AIGs are boolean circuits composed solely of two-input AND gates and inverters. In contrast to BDDs [17], 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.

AIGs enjoy a widespread application in several areas of formal verification, e. g. in combinational equivalence checking [18]. In [16] FRAIGs were tailored towards the representation and manipulation of sets of states in symbolic model checking and replaced BDDs as a compact representation of large discrete state spaces.

As we will show in section III-D, AIGs provide all necessary operations to be used as a symbolic data structure for QBF solving.

III. Our Approach

We divide our approach into four phases: (1) The *preprocessing* phase, in which techniques are applied to simplify the input QCNF formula (most of them are known from literature) (2) the *structure extraction* phase, where functional definitions of variables are extracted from the QCNF resulting in a structural, prenex QBF, (3) the *early quantification* phase, where a non-linear quantifier structure is extracted from the QBF yielding a tree-shaped general QBF formula, and (4) the *symbolic quantifier elimination* phase, in which the QBF formula is translated to a symbolic, AIG based representation, allowing the application of AIG operations to eliminate the quantifiers present in the formula.

A. Preprocessing

As shown in [19] simple QCNF preprocessing techniques may boost the overall solving performance when being invoked prior to the actual QBF solver. We apply the following preprocessing steps on the QCNF instance until we reach a fixpoint:

Unit clause propagation [20], *universal reduction* [19], elimination of *pure literals* [21], *equivalence reduction* [22], [8] based on the detection of strongly connected components (SCCs), removal of clauses subsumed [8] by binary clauses, and finally, detection of *implication chains* of the form $x \rightarrow \dots \rightarrow \overline{x}$ in the binary clause graph, resulting in a new unit literal $\overline{x}$.

Each of these simplification rules either detects (un)satisfiability of the whole formula, leaves the formula untouched or reduces the size of the QCNF. Directly after preprocessing we check the resulting QCNF formula for *trivial truth* and *trivial falsity* [21].

B. Structure Extraction

Often, QBF formulas are generated from circuit descriptions such as netlists of standard logic gates. A commonly used translation technique is the *Tseitin encoding* [23] that introduces propositional variables for each circuit signal and establishes the functional dependencies between the signal variables by encoding the gates' characteristic logic functions by sets of clauses. For example an n-input AND-gate $y \leftrightarrow AND(x_1, \dots, x_n)$ has a Tseitin encoding consisting of $n+1$ clauses: $(\overline{y} \vee x_1), \dots, (\overline{y} \vee x_n), (y \vee \overline{x}_1 \vee \dots \vee \overline{x}_n)$.

Considering a functional definition $y \leftrightarrow f(x_1, \dots, x_n)$, we call y the *defined variable*, the function f is referred to as the *definition* of y, and the set of clauses establishing the functional definition is called the set of *definition clauses*.

Our aim is to extract definition clauses from a QBF instance (irrespective of the question whether they were really introduced originally by Tseitin transformation or not) and build a structural, non-clausal QBF formula by (1) substituting the defined variables with structural representations of their definitions, (2) removing the clauses representing the functional definition from the CNF, and (3) removing the defined variable from the quantifier prefix.

In a later step, this structural representation is directly translated to a symbolic representation based on AIGs.

Several works considered the detection and exploitation of definitions in CNF preprocessing for propositional SAT (e. g. [8]). The authors make use of found definitions by substituting them into the CNF, and then applying the distributive law to produce a flat CNF representation. During this distribution step, the CNF's size may increase.

In QBF reasoning the exploitation of functional definitions is applied in [6] and [10]. In both works, definitions are used to eliminate variables from the CNF representing the QBF formula. Like in the case of SAT the CNF's size may increase in this step.

We differ from these previous works in the following substantial aspect: we are using a representation based on AIGs instead of CNF, which lets us apply structural substitution. An increase in size by applying the law of distributivity is avoided in our approach.

In contrast to SAT we have to consider the quantifier level and quantifier type (existential or universal) of variables which occur in functional definitions. Functional definitions establish a natural order on the involved variables, since the defined (output) variable functionally depends on the definition's inputs: If we assign values to the definition's inputs first, then a unique assignment to the output variable is implied. If the quantifier prefix of the QBF respects this order, i.e., if the quantification levels of the definition's inputs are smaller or equal to the level of the defined variable, we call the quantifier order *consistent* with the functional definition, otherwise it is called inconsistent.

We now have a look at all four possible combinations of order consistency and the type of the defined variable's quantifier, and formally investigate whether a substitution is possible, and what the result of a substitution is:

1) If the quantifier order is consistent with the functional definition, and the defined variable is existential, then the substitution is sound. Theorem 1 gives a formal proof.
2) If the quantifier order is inconsistent with the functional definition, and the defined variable is existential, the substitution is *not* sound, as one can see by the following counter example: $\exists z \forall x \exists a \forall y.(z \vee a) \wedge (a \leftrightarrow x \wedge y)$ is unsatisfiable, already $\exists z \forall x \exists a \forall y.(a \leftrightarrow x \wedge y)$ is unsatisfiable. Substituting a by $x \wedge y$, results in $\exists z \forall x \forall y.(z \vee (x \wedge y))$, which is satisfiable.
3) If the quantifier order is consistent with the functional definition, and the defined variable is universal, the substitution is sound, but then the instance is unsatisfiable, as can be shown by a simple variation of Theorem 1.
4) If the quantifier order is inconsistent with the definition, and the defined variable is universal, the substitution is *not* sound, as we again can see by a simple counter example: Consider the formula $\forall a \exists x \exists y.(a \leftrightarrow x \wedge y) \wedge a$. It is unsatisfiable, since it contains the universal unit literal a, but the substituted instance $\exists x \exists y.(x \wedge y)$ is satisfiable.

Theorem 1 (Substitution of Existential Definitions) *Let $Q_1 \exists y Q_2.\phi$ be a prenex QBF, where Q_1 and Q_2 are sequences of quantifiers, $a_1, \ldots, a_l$ are the variables quantified in Q_1, and $b_1, \ldots, b_k$ are the variables quantified in Q_2, $\{a_1, \ldots, a_l\} \cap \{b_1, \ldots, b_k\} = \emptyset$. Let ϕ be a conjunction of a boolean formula $\phi'(a_1, \ldots, a_l, y, b_1, \ldots, b_k)$ and the definition clauses G_y for $y \leftrightarrow f(x_1, \ldots, x_n)$, let $\lambda(x_i) \leq \lambda(y)$ hold for all $i \in \{1, \ldots, n\}$. Then*

$$Q_1 \exists y Q_2.G_y \wedge \phi'(a_1, \ldots, a_l, y, b_1, \ldots, b_k)$$
is equivalent to
$$Q_1 Q_2.\phi'(a_1, \ldots, a_l, f(x_1, \ldots, x_n), b_1, \ldots, b_k)$$
w. r. t. satisfiability.

Proof: According to the assumptions $\{x_1, \ldots, x_n\} \subseteq \{a_1, \ldots, a_l\}$ holds. Since G_y does not depend on any variables of Q_2, it can be pushed outwards until $\exists y$, resulting in $Q_1 \exists y.(G_y \wedge (Q_2.\phi'(a_1, \ldots, a_n, y, b_1, \ldots, b_k)))$. Replacing G_y by the equivalent characteristic function $y \leftrightarrow f(x_1, \ldots, x_n)$ yields $Q_1 \exists y.[(y \leftrightarrow f(x_1, \ldots, x_n)) \wedge (Q_2.\phi'(a_1, \ldots, a_l, y, b_1, \ldots, b_k))]$. Now we eliminate the existential quantifier by cofactoring and obtain the result $Q_1.F$ with $F = \{[f(x_1, \ldots, x_n) \wedge (Q_2.\phi'(a_1, \ldots, a_l, 1, b_1, \ldots, b_k))] \vee [\overline{f(x_1, \ldots, x_n)} \wedge (Q_2.\phi'(a_1, \ldots, a_l, 0, b_1, \ldots, b_k))]\}$.

For an arbitrary, but fixed valuation of $a_1, \ldots, a_l$ (and thus $x_1, \ldots, x_n$) $f(x_1, \ldots, x_n)$ also evaluates to a fixed value. If $f(x_1, \ldots, x_n)$ evaluates to 1, F can be simplified to $(Q_2.\phi'(a_1, \ldots, a_l, 1, b_1, \ldots, b_k))$, if it evaluates to 0, it can be simplified to $(Q_2.\phi'(a_1, \ldots, a_l, 0, b_1, \ldots, b_k))$. Thus we can equivalently transform $Q_1.F$ to $Q_1 Q_2.\phi'(a_1, \ldots, a_l, f(x_1, \ldots, x_n), b_1, \ldots, b_k)$, which is the desired result. ■

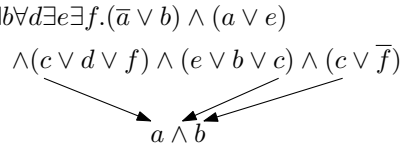
In our implementation, the definition detection routine is able to find definitions for (multi-input) AND gates, (multi-input) OR gates, and two-input XOR-gates, all with arbitrarily negated inputs. If multiple definitions for one variable are found, we extract the one that depends on the most clauses. The extraction of functional definitions leading to cyclic dependencies of variables is omitted.

Notice, that we do not transform the result of a substitution back to QCNF format by distribution. Instead we obtain a QCNF where a subset of the variables is associated with substitution instances. Iterative applications of substitutions lead to substitution instances with a graph-like structure just as a boolean circuit.

Example 1 *As an example, consider the following QCNF:*

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

We detect a definition for c: $c \leftrightarrow a \wedge b$ using the definition clauses $(\overline{c} \vee a)$, $(\overline{c} \vee b)$, and $(c \vee \overline{a} \vee \overline{b})$. According to Theorem 1, we remove the definition clauses and structurally replace c by $a \wedge b$, leading to a representation like this:



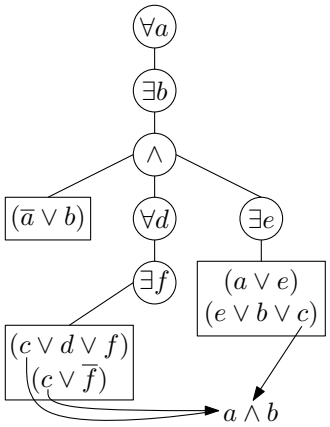
The connections from c to $a \wedge b$ denote the association between the variable and its definition.

C. Early Quantification

In [5] the authors propose a method to construct a tree-shaped quantifier structure from a QCNF instance with linear quantifier prefix, and show how to benefit from the structure in the solving phase. We modified the quantifier tree algorithm from [5] to be able to cope with our hybrid QBF formulas by making it aware of functional definitions for CNF variables.

The result of the algorithm is a generalized *quantifier tree*. Internal nodes are either *quantifier nodes* labelled with a quantifier and a variable or *conjunctive nodes*, the leaves contain sets of clauses with variables which may be associated to definitions. The order of quantifiers along each path from the root to a leaf is consistent with the original quantifier order.

Example 2 *The following (generalized) quantifier tree results for the QBF from Example 1:*



The variable c is linked to its definition $a \wedge b$.

D. Symbolic Quantifier Elimination

Given a QBF formula in form of a quantifier tree with definitions, we use the AIG package from [16] to recursively create AIGs for the formula's nodes starting at the tree's leaves.

The standard boolean operations (conjunction, disjunction, negation, XOR) occurring in the structural QBF formula can all be broken down to conjunction and negation, which can be carried out on AIGs by adding AND nodes and inverters.

Quantifier elimination on AIGs is performed by cofactoring, i. e.$\exists x.f = f|_{x=0} \vee f|_{x=1}$, where f is a boolean function represented by an AIG, $f|_{x=0}$ is the negative cofactor of f w. r. t. the variable x, and $f|_{x=1}$ is the positive cofactor. The cofactors themselves are computed by structurally copying the original AIG, while replacing the quantified variable by a constant value. Quantifier elimination may in the worst case double the AIGs size, an effect that is tackled by various methods: (1) merging of equivalent nodes, (2) quantifier scheduling, and (3) BDD based structure compaction.

While creating AIG nodes we maintain the functional reduction property [15], i. e. all nodes must represent unique boolean functions, by merging equivalent nodes. Equivalent nodes are found in a three staged process: first we apply structural hashing to detect isomorphic nodes, secondly we select a set of candidate nodes for equivalence by functional simulation, the third step uses SAT to prove/disprove equivalence with each candidate node. In case of an equivalence, the nodes are immediately merged. Especially when eliminating quantifiers, the functional reduction property leads to a significant reduction of size, since equivalent substructures of both cofactors are merged.

Quantifier scheduling [16] is a heuristic approach, that is used if a series of quantifiers with the same level is to be eliminated: the size of each quantifier elimination result is estimated, and the quantifier leading to the smallest increase in size is eliminated first.

BDD sweeping [16] is used to 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, an structurally equivalent AIG is built. Finally, the original AIG is replaced by the equivalent, but smaller new AIG.

Another method for AIG compaction is AIG rewriting [24], that iteratively replaces AIG subgraphs by smaller, precomputed subgraphs, preserving their functionality and decreasing the overall size of the AIG.

In our implementation, we first create AIG input nodes for the QBF variables without definitions, then we recursively create AIGs for the QBF variables that have functional definitions. Finally, we apply Algorithm 1 *CreateAIG* to the root of the quantifier tree, which traverses the tree in a depth-first manner and creates AIGs for each node.

```
Input  : Quantifier Tree node n
Output: AIG for n
begin
    if n is a conjunctive node then
        AIG t := ⋀_{c∈children(n)} CreateAIG(c);
        return t
    else if n is a universal node then
        AIG t := CreateAIG(child(n));
        return UniversalQ(t, variable of n)
    else if n is a existential node then
        AIG t := CreateAIG(child(n));
        return ExistentialQ(t, variable of n)
    else if n is clause node then
        AIG t := ⋀_{c∈clauses(n)} CreateClauseAIG(c);
        return t
end
```

Algorithm 1: CreateAIG

The subroutine *CreateClauseAIG* creates an AIG for a given clause by computing the disjunction of the AIGs of the clause's literals. Note that the AIG for a literal may be different

from a variable or negated variable in case that the literal corresponds to a variable that has a definition. *UniversalQ* and *ExistentialQ* are the universal and existential quantifier elimination routines of the AIG package, "$\bigwedge$" denotes the AND operation on AIGs. After each quantifier elimination, we try to compress the resulting representation using BDD sweeping and AIG rewriting. When *CreateAIG* terminates, an AIG is present for the root node. If this AIG is equivalent to 0[1] the QBF is unsatisfiable, otherwise it is satisfiable.

IV. Experimental Results

Based on the methods presented in this paper we implemented a prototype QBF solver. In order to evaluate the overall performance of our approach, we compared our solver with the latest, publicly available versions of the following state-of-the-art QBF solvers: Quantor v3.0 [6], QuBE 6.4 [25], and sKizzo v0.8.2 [5]. This selection of solvers covers a broad spectrum of decision techniques for QBF: Quantor uses resolution and expansion based variable elimination, QuBE is a search based solver, sKizzo applies quantifier tree reconstruction, symbolic skolemization, and propositional ground reasoning. The experiments were performed for the complete QBFEVAL'07[26] benchmark set. All experiments were run on a dual AMD dual-core Opteron machine with 2.8 GHz and 16 GB of memory using a time limit of 600 seconds and a memory limit of 4 GB.

Table I shows the results for different benchmark families. Column *Family* lists the names of the benchmark families, column *Inst.* lists the number of instances of each family. For each solver the columns *S* show the numbers of instances of each family where the solver succeeded, and the column *Time* lists the total solving time in CPU seconds for each instance[2]. For our AIG based solver AIGsolve, we also listed the number of instances that have been solved during the preprocessing phase according to Sect. III-A (column *P*), and the total percentages of variables for which functional *definitions* have been extracted (column *D*), furthermore column *U* lists the number of instances that AIGsolve was able to solve uniquely among the tested solvers, column *H7* (*H8*) lists the number of hard instances, according to QBFEVAL'07 (QBFEVAL'08), on which our solver succeeded. To support our claim that the extraction of functional definitions is beneficial for the solving algorithm, we also ran our solver with extraction of definitions *turned off*. The group of columns labelled *AIGsolve/nodef* lists the numbers of solved instances and the solving times for this variant.

We can solve 157 out of 1136 instances just by preprocessing and trivial satisfiability checks. The structure extraction part of our approach is able to detect and extract functional definitions for a total of 51.9% of the variables remaining after preprocessing. Moreover, there are families where functional definitions can be found for more than 90% of the variables: *bbox-01x-qbf*, *C+term1*, *tipdiam*, *tipfixpoint*. This result lets us conclude that most QBF benchmarks can significantly be simplified by preprocessing and that there is a fair amount of structure from which a QBF solver can profit during the actual solving phase.

In total, our solver succeeds on 537 instances, a result that clearly outperforms Quantor (422 solved instances), while being slightly above the number of instances solved by sKizzo (533 solved instances). Only QuBE, which currently is the fastest solver according to QBFEVAL, is able to outperform our approach with respect to the number of solved instances (614 solved instances). Taking a closer look at the experimental results, we observe that our solver is able to uniquely solve 53 instances among the tested solvers, moreover it is successful on 48 (51) instances that are marked as *hard* in the QBFEVAL'07 (QBFEVAL'08), i. e. instances that no solver participating in the competitive evaluations could solve within the time limit of 600 seconds. Most of these instances are from the *tipdiam* and *tipfixpoint* families, where our structure extraction method detects a large number of definitions, and thus optimally supports the solver's AIG core. Without the exploitation of functional definitions, our solver is only able to solve 401 instances. This result clearly shows that extracting functional definitions indeed is beneficial for our solver.

Our experimental results not only show the effectiveness of our approach in general, but they also show that our solver has strengths which are complementary to those of search based solvers like QuBE. This gives a strong motivation for a future integration of both concepts in a combined approach.

V. Conclusions

In this paper, we propose to extract structural information from QBF formulas in form of functional definitions of variables, and tree-shaped quantifier structures. This information is directly used to build a symbolic, AIG based representation of the QBF formula, on which AIG operations are applied to eliminate the quantifiers. Based on these ideas a prototype QBF solver was implemented and evaluated on the complete QBFEVAL'07 benchmark set. The experimental results show that indeed a lot of structure can be extracted from the benchmarks, allowing the prototype solver to compete successfully with other state-of-the-art solvers, and to outperform the best solvers by a number of – until then – unsolved instances. This clearly demonstrates that the proposed methods are relevant for QBF solving and that they are able to successfully complement the decision procedures implemented by other QBF solvers.

[1] Due to the functional reduction property, this equivalence is automatically detected when creating the AIG for the root node

[2] Failed instances (i. e. the solver violated the memory or time limit) are considered to contribute 600 CPU seconds (the time limit)

References

[1] C. Scholl and B. Becker, "Checking Equivalence for Partial Implementations," in *Proc. of the 38th Design Automation Conf. (DAC 2001)*. ACM, 2001, pp. 238–243.

[2] M. Herbstritt and B. Becker, "On Combining 01X-Logic and QBF," in *Proc. of the 11th Int. Conf. on Computer Aided Systems Theory (EUROCAST 2007)*, 2007, pp. 531–538.

TABLE I
COMPARISON WITH OTHER SOLVERS

		AIGsolve							AIGsolve/nodef		Quantor		sKizzo		QuBE	
Family	Inst.	P	D	S	Time	U	H7	H8	S	Time	S	Time	S	Time	S	Time
Adder	11	0	0.0%	9	2268	1	0	0	9	2260	4	4248	8	2143	1	6001
bbox-01X-qbf	450	104	96.5%	173	169906	0	0	1	147	184492	130	192018	210	152761	290	100180
bbox_design	28	0	89.0%	23	3790	0	0	0	18	11682	0	16800	1	16456	28	15
Blocks	6	0	36.8%	0	3600	0	0	0	0	3600	6	79	3	2045	2	2666
BMC	132	16	59.8%	30	62075	0	0	0	36	49429	113	13348	105	17954	56	47365
C+term1	12	0	92.8%	9	2328	0	1	0	4	5179	6	4195	5	4464	8	3039
Chain	2	2	-	2	0	0	0	0	2	0	2	0	2	0	2	0
conf_planning	24	0	56.8%	1	13802	0	0	0	2	13207	14	6801	8	10029	6	11229
Connect4	1	0	0.9%	0	600	0	0	0	0	600	0	600	0	600	0	600
Counter	3	0	3.2%	2	904	0	0	0	3	77	3	15	3	5	0	1800
Debug	38	0	2.6%	0	22800	0	0	0	0	22800	29	8922	9	19233	0	22800
evader-pursuer	4	0	3.4%	0	2400	0	0	0	0	2400	0	2400	1	1802	0	2400
FPGA	1	0	0.0%	0	600	0	0	0	1	286	1	3	1	2	1	78
jmc_quant	3	0	42.9%	2	1105	0	0	0	2	625	0	1800	1	1716	3	11
k	61	0	51.5%	56	4070	0	0	0	57	3207	38	13948	61	1479	23	24627
MutexP+Qshifter	8	0	0.0%	8	566	0	0	0	8	495	4	2417	8	4	3	3045
s	10	0	72.5%	10	229	0	0	0	9	1666	10	480	10	133	10	6
Sort_networks	53	0	12.1%	0	31800	0	0	0	7	29575	27	16042	18	21853	19	21636
SzymanskiP	6	0	0.0%	0	3600	0	0	0	0	3600	0	3600	0	3600	6	134
tipdiam	85	0	96.0%	77	6556	16	15	19	38	32398	24	36699	50	24729	57	17887
tipfixpoint	196	35	95.2%	133	42138	36	32	31	56	88502	9	112333	27	102532	99	61620
Toilet	2	0	60.9%	2	137	0	0	0	2	145	2	1	2	1	0	1200
Total	1136	157	51.9%	537	375274	53	48	51	401	456225	422	436749	533	383541	614	328338

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

[4] U. Egly, T. Eiter, H. Tompits, and S. Woltran, "Solving Advanced Reasoning Tasks Using Quantified Boolean Formulas," in *Proc. of the 17th Nat. Conf. on Artificial Intelligence and 12th Conf. on Innovative Applications of Artificial Intelligence (AAAI/IAAI)*, 2000, pp. 417–422.

[5] M. Benedetti, "Quantifier Trees for QBFs," in *Proc. of the 8th Int. Conf. on Theory and Applications of Satisfiability Testing (SAT 2005)*, 2005, pp. 378–385.

[6] A. Biere, "Resolve and Expand," in *Proc. of Seventh Int. Conf. on Theory and Applications of Satisfiability Testing (SAT 2004), Selected Papers*, 2004, pp. 59–70.

[7] A. Kuehlmann, M. K. Ganai, and V. Paruthi, "Circuit-based Boolean Reasoning," in *Proc. of the 38th Design Automation Conf. (DAC 2001)*, 2001, pp. 232–237.

[8] N. Eén and A. Biere, "Effective Preprocessing in SAT Through Variable and Clause Elimination," in *Proc. of the 8th Int. Conf. on Theory and Applications of Satisfiability Testing (SAT 2005)*, 2005, pp. 61–75.

[9] R. Ostrowski, É. Grégoire, B. Mazure, and L. Sais, "Recovering and Exploiting Structural Knowledge from CNF Formulas," in *Proc. of the 8th Int. Conf. on Principles and Practice of Constraint Programming (CP 2002)*, 2002, pp. 185–199.

[10] E. Giunchiglia, P. Marin, and M. Narizzano, "sQueezBF: An Effective Preprocessor for QBF," in *Proc. of the 2nd Int. Workshop on Quantification in Constraint Programming (QiCP 2008)*, 2008.

[11] L. Zhang, "Solving QBF with Combined Conjunctive and Disjunctive Normal Form," in *Proc. of the 21st Nat. Conf. on Artificial Intelligence (AAAI 2006)*, 2006.

[12] F. Lonsing and A. Biere, "Nenofex: Expanding NNF for QBF Solving," in *Proc. of the 11th Int. Conf. on Theory and Applications of Satisfiability Testing (SAT 2008)*, 2008.

[13] M. Benedetti, "Evaluating QBFs via Symbolic Skolemization," in *Proc. of the 11th Int. Conf. on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR04)*, ser. LNCS. Springer, 2005, no. 3452.

[14] E. Giunchiglia, M. Narizzano, and A. Tacchella, "Quantifier Structure in Search-Based Procedures for QBFs," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 26, no. 3, pp. 497–507, 2007.

[15] A. Mishchenko, S. Chatterjee, R. Jiang, and R. K. Brayton, "FRAIGs: A unifying representation for logic synthesis and verification," EECS Dept., UC Berkeley, Tech. Rep., 03 2005.

[16] F. Pigorsch, C. Scholl, and S. Disch, "Advanced Unbounded Model Checking Based on AIGs, BDD Sweeping, And Quantifier Scheduling," in *Proc. of the Conf. on Formal Methods in Computer Aided Design (FMCAD 2006)*. IEEE Computer Society Press, Nov 2006, pp. 89 – 96.

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

[18] A. Mishchenko, S. Chatterjee, R. K. Brayton, and N. Eén, "Improvements to combinational equivalence checking," in *Proc. of the Int. Conf. on Computer-Aided Design (ICCAD 2006)*, 2006, pp. 836–843.

[19] H. Samulowitz, J. Davies, and F. Bacchus, "Preprocessing QBF," in *Proc. of the 12th Int. Conf. on Principles and Practice of Constraint Programming (CP 2006)*, 2006, pp. 514–529.

[20] M. Davis and H. Putnam, "A Computing Procedure for Quantification Theory," *Journal of the ACM*, vol. 7, no. 3, pp. 201–215, 1960.

[21] M. Cadoli, M. Schaerf, A. Giovanardi, and M. Giovanardi, "An algorithm to evaluate quantified boolean formulae," in *Journal of Automated Reasoning*. AAAI Press, 1998, pp. 262–267.

[22] H. Samulowitz and F. Bacchus, "Binary Clause Reasoning in QBF," in *Proc. of the 9th Int. Conf. on Theory and Applications of Satisfiability Testing (SAT 2006)*, 2006, pp. 353–367.

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

[24] A. Mishchenko, S. Chatterjee, and R. Brayton, "DAG-aware AIG rewriting a fresh look at combinational logic synthesis," in *Proc. of the 43rd annual Conf. on Design Automation (DAC 2006)*. ACM, 2006, pp. 532–535.

[25] E. Giunchiglia, M. Narizzano, and A. Tacchella, "QUBE: A System for Deciding Quantified Boolean Formulas Satisfiability," in *Proc. of the 1st Int. Joint Conf. Automated Reasoning (IJCAR 2001)*, 2001, pp. 364–369.

[26] ——, "QBF Evaluation 2007," http://www.qbflib.org/index_eval.php.