Using Autarky to Evaluate Quantified Boolean Formulae

Jens Rühmkorf
Simulation and Software Technology
German Aerospace Center (DLR)
Linder Höhe, D-51147 Köln, Germany
E-mail: Jens.Ruehmkorf@dlr.de

***Abstract*— In this paper, we discuss algorithmical implications for the extension of autarky from propositional logic to evaluate quantified boolean formulae (QBF). First, the Davis-Putnam procedure for the satisfiability problem (SAT) is described. Then we explain efficient known data structures for SAT and extensions to QBF which we used in our solver. Finally, we introduce the concept of autarky and describe how detecting 2-autarky structures in a given QBF formula helps pruning the search tree. To the best of our knowledge we are the first to describe such techniques for QBF.**

***Keywords*—Autarky; Davis-Putnam; SAT; QBF**

I. INTRODUCTION

In recent years, the language of quantified boolean formulae (QBF) has gained importance for practical applications. QBF allows for a concise representation of many classes of problems [1], [2]: Gopalakrishnan et al. study the problem of formally verifying shared memory multiprocessor executions against memory consistency models for the Intel Itanium by translating occurring problems to the satisfiability problem (SAT) and QBF [1]. Mneimneh et al. also consider the application area of formal hardware verification and transform the diameter problem — determining the length of a longest of all shortest paths — for a class of large digraphs to QBF, but end up converting their problem to SAT, because no existing QBF solver is able to solve their problems [2].

The paper is organized as follows: Section II introduces preliminary definitions and concepts. Section III describes the Davis-Putnam procedure for SAT along with efficient data structures. Section IV discusses efficient data structures used for extending Davis-Putnam to QBF. Based on our experimental results (due to space limitations not discussed in this paper). Sections V and VI derive enhancements to the current algorithm by extending the concept of autarky to QBF and discuss possible implementation approaches. Section VII concludes the paper by briefly evaluating the current status and provides information on future development.

II. PRELIMINARIES

A quantified boolean formula $\Phi = q_1 z_1 \cdots q_n z_n \ \varphi$ consists of a sequence of quantified variables, the so-called *prefix*, followed by a quantifier free SAT formula φ, the *matrix* of the formula. The prefix $q_1 z_1 \cdots q_n z_n$ contains universal $\forall$ and existential $\exists$ quantifiers for propositional variables z_i occurring in φ. The *evaluation problem* for a given QBF Φ is to decide whether Φ is true or not. Example QBF formulas are $\forall y_1 \exists x_2 \ (y_1 \vee \neg x_2) \wedge (\neg y_1 \vee x_2)$ which is true and $\forall y_1 \forall y_2 \ (y_1 \vee y_2)$ which is false.

A *literal* L is a variable z or $\neg z$. For two different literals L_i, L_j with corresponding variables z_i, z_j of a QBF formula Φ we may write $L_i < L_j$ if z_i occurs to the left of z_j in the prefix of Φ. We use $\mathrm{Lit}(Z)$ as shorthand notation for the set of literals for a given set of variables Z. Similarly, $\mathrm{Var}(\Phi)$ and $\mathrm{Var}(\varphi)$ are used for the variable sets occurring in a QBF formula Φ and a SAT formula φ, respectively. A *clause* is a formula $\kappa = (L_1 \vee \cdots \vee L_k)$ with literals L_i; the second clause of the first example is $(\neg y_1 \vee x_2)$ with literals $\neg y_1$ and x_2. We say a SAT formula φ is in *conjunctive normal form* (CNF), if it is expressed by a conjunction of clauses $\varphi = \kappa_1 \wedge \cdots \wedge \kappa_\ell$.

III. DAVIS-PUTNAM FOR SAT

Both SAT and QBF describe prototypical complete problems for the important complexity classes $\mathcal{NP}$ and $\mathcal{PSPACE}$, respectively. Syntactically restricted forms of QBF describe complete problems for $\Sigma_k^{\mathcal{P}}$ and $\Pi_k^{\mathcal{P}}$ within the polynomial time hierarchy $\mathcal{PH}$. For the remainder of this paper, we consider only quantified boolean formulae Φ whose matrix φ is in CNF. Indeed, for a given QBF formula Φ we may generate in linear time an equivalent quantified boolean formula whose matrix is in CNF [4, ch. 7].

The evaluation algorithm used within this paper is a generalization of the Davis-Putnam procedure for SAT [5] to QBF [6]. Figure 1 on the following page describes the Davis-Putnam algorithm as recursive function. The algorithm utilizes the following two fundamental observations:

Lemma 1 (monotone literal [5]) *If the literal L is* monotone, *i.e. by definition, L occurrs only positive ($L = x$) or negative ($L = \neg x$) within the* CNF *formula α, then α is equivalent to $\alpha[L/1]$ or $\alpha[L/0]$, respectively.* □

Lemma 2 (unit clause [5]) *Let L be the literal of a unit clause of the* CNF *formula α. Then α is satisfiable if and only if $\alpha[L/1]$ is satisfiable.* □

Here $\alpha[L/\epsilon]$ denotes the formula obtained from α by replacing each occurrence of L with $\epsilon \in \{0, 1\}$. Furthermore,

 ISBN: 978-1-61208-101-4

a clause κ is called k-clause, if κ contains only k literals $(L_1 \vee \cdots \vee L_k)$, a 1-clause (L) is called *unit clause*.

Function *boolean* `Davis-Putnam(`*CNF formula** α`)`

Input: Pointer to CNF formula α.
Output: `true` if α is satisfiable and `false` otherwise.

```
begin
    if α = 1 then return true;
    if α = 0 then return false;
    L ← Pure-Literal(α);
    if L ≠ NULL then
        return Davis-Putnam(α[L/1]);
    L ← Unit-Literal(α);
    if L ≠ NULL then
        return Davis-Putnam(α[L/1]);
    L ← Choose-Literal(α);
    if Davis-Putnam(α[L/1]) then
        return true;
    return Davis-Putnam(α[L/0]);
end
```

Figure 1: The Davis-Putnam algorithm for SAT.

Function `Pure-Literal` corresponds to lemma 1: it returns for a formula α a pointer to a monotone literal if it exists and `NULL` otherwise. Function `Unit-Literal` utilizes lemma 2 and returns a unit clause if it exists and otherwise `NULL`. The function `Choose-Literal` defines the heuristic which literal to use next for branching. Good experimental results are obtained by using the lexicographical heuristic [7]: Let $h_i(L)$ be the number of clauses of length i in which a given literal L occurs. Then calculate:

$$H_i(A) = \max\big(h_i(x), h_i(\neg x)\big) + 2\min\big(h_i(x), h_i(\neg x)\big) \quad (1)$$

Then, the variable with maximal vector $(H_1(x), \cdots, H_n(x))$ according to the lexicographical ordering is chosen.

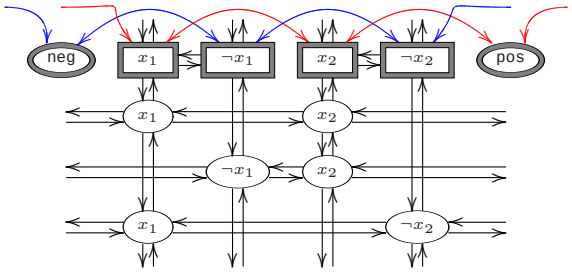


Figure 2: Data structure for the 2-CNF formula $\alpha_1 = (x_1 \vee x_2) \wedge (\neg x_1 \vee x_2) \wedge (x_1 \vee \neg x_2)$.

As can be seen in figure 1, Davis-Putnam uses a depth first strategy where backtracking occurs when a leaf labelled with 0 is reached in its execution tree. The algorithm executes a method call `Assign(`α, L`)` or `Assign(`α, $\neg L$`)` for every occurrence of $\alpha[L/1]$ or $\alpha[L/0]$, respectively. Upon leaving the recursion on level $\alpha[L/1]$ the implicitly called method `Unassign(`α, L`)` modifies the current formula to α by making use of a recursion stack.

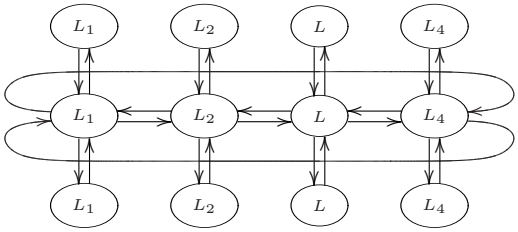


Figure 3: Data structure of clause $\kappa_L = (L_1 \vee L_2 \vee L \vee L_4)$.

The formulation of Davis-Putnam is surprisingly simple. Of key importance is the realization of data structures that efficiently support necessary operations. With the sparse data structure by Böhm and Speckenmeyer [7] used for our solver the operations `Assign()` and `Unassign()` need time $\mathcal{O}\big(|\alpha| - |\alpha[L/1]|\big)$, the test for unit clauses needs time $\mathcal{O}(1)$.

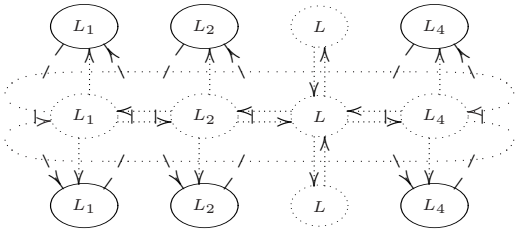


Figure 4: Remove clause κ_L in time $\mathcal{O}(|\kappa_L|)$.

Figure 2 shows the used data structure for a 2-CNF formula. There, literals and clauses are connected in the following way: Every occurrence of a literal in a clause corresponds to a literal object within the data structure, in figures 2 to 5 depicted by $\neg x$ or L.

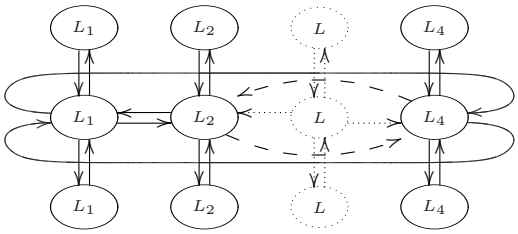


Figure 5: Shorten clause κ_L in time $\mathcal{O}(1)$.

All literals of a clause are connected through a doubly-linked circular clause list as shown in figure 3. All literals of the same type are connected through doubly-linked circular lists, so called *literal occurrence lists* depicted by the columns in figure 2. The head of such a literal occurrence list is displayed by $\neg x$. These list heads are themselves divided in two doubly-linked, circular lists. The list pos

connects the list heads of all positive literals, whereas the list 

connects the list heads of all negative literals. These two lists represent the yet unassigned literals in the given formula.

When applying changes within the data structure, bookmarking links are set to be able to easily revert changes made when traversing the execution tree. Figure 4 on the preceding page shows the changes to the datastructure when removing a clause (e.g. when performing $\alpha[L/1]$) and similarly figure 5 shows the changes performed when shortening a clause (e.g. when performing $\alpha[L/0]$).

Operation	Runtime
`Unassign(`α`,`L`)`, `Assign(`α`,` L`)`	$\mathcal{O}(\lvert\alpha\rvert - \lvert\alpha[L/1]\rvert)$
`Unit-Literal(`α`)`	$\mathcal{O}(1)$
Remove clause κ from α	$\mathcal{O}(\lvert\kappa\rvert)$
Remove L from clause κ	$\mathcal{O}(1)$
Find clause κ with $L \in \kappa$	$\mathcal{O}(1)$

Figure 6: Runtime of operations for CNF data structure.

Figure 6 lists important operations of the data structure with their corresponding runtime behaviour.

IV. Evaluating QBF

This section describes the changes necessary to extend Davis-Putnam to QBF. The following lemma proves to be an easy but fundamental tool for this:

Proposition 3 (substitution lemma [4]) *Let Π be a prefix and let Φ_1 as well as Φ_2 be two quantified boolean formulae. Then follows from the equivalence of Φ_1 and Φ_2, written $\Phi_1 \approx \Phi_2$, that the quantified formula $\Pi\ \Phi_1$ is equivalent to $\Pi\ \Phi_2$, in other words: $\Pi\ \Phi_1 \approx \Pi\ \Phi_2$.* □

According to proposition 3, we may transform the matrix of a QBF formula, just like we would for a CNF formula. The proof is an easy induction on the length of the prefix Π. Furthermore, we consider the following two lemmas which may be easily derived from their CNF counterparts.

Lemma 4 (monotone quantified literal [6]) *Let Φ be a quantified boolean formula and let L be a monotone literal of Φ, i.e. a literal whose complement $\neg L$ does not occur in the matrix of Φ. Then the following holds:*

1) *In case L is $\exists$-quantified, then Φ is true if and only if $\Phi[L/1]$ is true.*
2) *In case L is $\forall$-quantified, then Φ is true if and only if $\Phi[L/0]$ is true.* □

Lemma 4 means that in case of an $\exists$-quantified monotone literal L we may remove all clauses containing L, whereas in case of an $\forall$-quantified such literal we may shorten all clauses that contain L by removing L.

We call a clause of a quantified boolean formula *unit existential clause* if it contains exactly one $\exists$-quantified literal for a variable y and all $\forall$-quantified literals for variables x occur to the right of y within the prefix of the formula. The QBF datastructure saves all literals of a clause in the order their corresponding variables occur within the prefix.

Lemma 5 (existential unit clause) *Let L be the $\exists$-quantified literal of an unit existential clause κ of a quantified boolean formula Φ, i.e., $L < L_i$ for all other ($\forall$-quantified) literals L_i of the clause κ. Then Φ is true if and only if $\Phi[L/1]$ is true.* □

Function *boolean* `DP-QBF(`*QBF formula** Φ`)`

Input: Pointer to QBF formula Φ.
Output: `true` if Φ evaluates to true and `false` otherwise.

```
begin
    if Φ = true or Φ = false then return Φ;
    L ← Pure-Literal(Φ);
    if L ≠ NULL then
        switch Quantifier(L) do
            case ∃ : return DP-QBF(Φ[L/1]);
            case ∀ : return DP-QBF(Φ[L/0]);
    L ← Unit-Literal(Φ);
    if L ≠ NULL then
        return DP-QBF(Φ[L/1]);
    L ← Choose-Literal(Φ);
    switch Quantifier(L) do
        case ∃ :
            if DP-QBF(Φ[L/1]) or DP-QBF(Φ[L/0])
            then
                return true;
            else
                return false;
        case ∀ :
            if DP-QBF(Φ[L/1]) and DP-QBF(Φ[L/0])
            then
                return true;
            else
                return false;
end
```

Figure 7: Skeleton of Davis-Putnam algorithm for QBF.

The Davis-Putnam extension to QBF may be formulated as described in figure 7. There the functions `Pure-Literal()` and `Unit-Literal()` correspond to lemmata 4 and 5, respectively. We examine the heuristic which delivers the literal to set next. Different to SAT the choice for QBF is restricted to the leftmost group of variables within the prefix that have the same quantifier. That means for a quantified boolean formula $\Phi = \forall Y_1 \exists X_2 \forall Y_3 \cdots \exists X_k\, \varphi$ with $\forall$-quantified variable sets Y_i and $\exists$-quantified sets X_j first all literals belonging to variables from Y_1 are considered, then all literals belonging to variables from X_2, and so forth.

 ISBN: 978-1-61208-101-4

The choice of a literal out of $\mathrm{Lit}(Y_i)$ respectively $\mathrm{Lit}(X_j)$ is then determined by the function `Choose-Literal()`. For the lexicographic heuristic for SAT described by equation (1), the literal is chosen which occurs most often in the shortest clauses of a given formula. Translated to QBF, a literal with such properties out of the leftmost prefix group is chosen. For QBF, the length of a clause is measured by counting the number of $\exists$-quantified literals whithin a clause, irrespective of its corresponding position within the clause (see [8] for a similar approach). For example: a clause $(x_1 \vee y_2 \vee y_3)$ is treated by this modified heuristic just like the clause $(y_4 \vee x_5 \vee y_6 \vee y_7)$, while the unmodified heuristic from equation (1) would rank the literals of the first clause better than literals from the second clause.

V. Utilizing Autarky for QBF

A function $\mathfrak{I} : \{x_0, x_1, x_2, \ldots\} \to \{0, 1\}$ for variables x_i is called a *truth assignment*. If $\mathfrak{I}$ is a *partial assignment* that operates on a subset of the variables $\mathrm{Var}(\varphi)$ of a SAT formula φ, then $\mathfrak{I}(\varphi)$ denotes the formula obtained by assigning truth values to this subset's variables accordingly.

Definition 6 (autark assignment [9]) *A truth assignment $\mathfrak{I}$ of some variables $\{x_{i_1}, \ldots, x_{i_k}\}$ of a* SAT *formula φ is called* autark, *if the following holds: every clause of φ that contains a variable x_{i_j} is already satisfied by $\mathfrak{I}$.*

If such an $\mathfrak{I}$ "touches" a clause of φ, this clause is already satisfied by $\mathfrak{I}$: every clause of $\mathfrak{I}(\varphi)$ occurs in φ. Autarky has the nice property that we may remove all clauses with variables x_{i_j} from φ without changing the satisfiability of φ. The following easy remark gives further insight:

Remark 7 (Satisfiability of autark assignments) If $\mathfrak{I}$ is an autark assignment of variables $V_{\mathrm{aut}} = \{x_{i_1}, \ldots, x_{i_k}\}$ for a SAT formula φ, then φ is satisfiable if and only if an assignment $\mathfrak{I}'$ exists that satisfies φ and the restriction of $\mathfrak{I}'$ to V_{aut} is identical to $\mathfrak{I}$, i.e., $\mathfrak{I}'|_{V_{\mathrm{aut}}} = \mathfrak{I}$.

Proof. Let $\mathfrak{I}$ be an autark assignemnt for φ, with variable set V_{aut} and let $\mathfrak{H}$ be a thruth assignment that satisfies φ. We may alter $\mathfrak{I}$ in accordance to $\mathfrak{H}$ by defining $\mathfrak{I}'(x) = \mathfrak{I}(x)$ if x belongs to V_{aut} and $\mathfrak{I}'(x) = \mathfrak{H}(x)$ otherwise. Then $\mathfrak{I}'$ satisfies φ. □

For the easy case of 1-autarky with $|V_{\mathrm{aut}}| = 1$ remark 7 corresponds to lemma 1 on page 1, the rule monotone literal. Our experiments showed that this rule lead to good results for quantified formulas, i.e. to considerably less branching nodes within our execution tree.

Therefore we examine how to extend the concept of autarky to quantified boolean formulae. We consider the case of 2-autarky for a QBF formula Φ. Without loss of generality Φ may contain no monotone literals. That means we examine all variable subsets from $\mathrm{Var}(\Phi)$ with size 2, respecting their order in the prefix. The idea is to compute these subsets in advance and later utilize them for search tree pruning. For this, the following cases need to be considered:

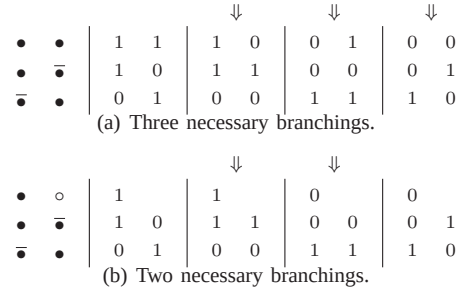
Case 1: 2-$\exists\exists$-autarky $\{x_{i_1}, x_{i_2}\}$. If an autark truth assignment exists for two $\exists$-quantified variables $x_{i_1} < x_{i_2}$, we have an already known case: all clauses, that contain either x_{i_1} oder x_{i_2} may be removed. This is also true if x_{i_1} and x_{i_2} do not belong to the leftmost prefix group.

Case 2: 2-$\forall\forall$-autarky $\{y_{i_1}, y_{i_2}\}$. For this case we closer examine the structure of all clauses that contain y_{i_1} or y_{i_2}. There are eight possibilites for membership of y_{i_1} or y_{i_2} within a clause. Figure 8(a) shows the four possible ways that a clause contains either y_{i_1} or y_{i_2}, positive or negative. Figure 8(b) shows the four possibilities that y_{i_1} as well as y_{i_2} are contained in a clause. There $\bullet$ describes a positive literal, $\bar{\bullet}$ describes a negative literal, and $\circ$ shows that the variable in question is not contained in the clause.

(a) 1-structure. (b) 2-structure

Figure 8: Clause structure for 2-autarky.

We have $2^8 = 256$ possible structural occurrences of two given distinct variables in the clauses of a CNF formula. Of these occurrences only those need to be considered where both variables occur at least once in a clause; so seven cases may be rejected. By a combinatorical argument with some case distinctions we may identify 90 cases of 2-autarkies and therefore 159 cases of not-2-autarkies — this includes symmetries and renamings of the kind $z \leftarrow \neg z$.



(a) Three necessary branchings.

(b) Two necessary branchings.

Figure 9: Branchings for 2-$\forall\forall$-autarky.

We discuss essential ideas with the help of some examples. Figure 9(a) shows a 2-autarky with three possible types of clause-structures $(\ldots y_{i_1} \vee y_{i_2} \ldots), (\ldots y_{i_1} \vee \neg y_{i_2} \ldots)$ as well as $(\ldots \neg y_{i_1} \vee y_{i_2} \ldots)$. First we consider the case that both variables are part of the leftmost prefix group. Then branching according to the first column (i.e. $\mathfrak{I}(y_{i_1}) = 1$ und $\mathfrak{I}(y_{i_2}) = 1$) does not make sense due to the 2-$\forall\forall$-structure.

Then, in the worst case, each of the other branchings needs to be considered.

How does the prefix order influence the algorithm? If y_{i_1} belongs to the leftmost prefix group we may branch immediately. Because we have 2-autarky, the assignment of y_{i_1} leads to a monotone literal y_{i_2} in the reduced formula (which may be pruned, column 2). If this branch does not lead to an abort, columns 3 and 4 need to be considered. Here we must wait with the assignment of y_{i_2} until either allowed by the prefix ordering or a special rule (monotone quantified literal, unit existential clause) applies.

For the case that y_{i_1} does not belong to the leftmost prefix group we may bookmark the 2-autarky for later consideration.

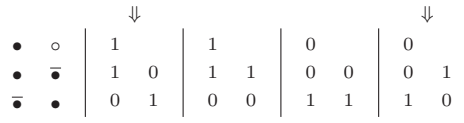


Figure 10: Two branchings for 2-∀∃-autarky.

For the case of figure 9(b) on the preceding page we have a different situation. Here also three different clause types need to be considered, but with only two meaningful branchings. Just like for figure 9(a) we do not need to branch as described in column 1. This also holds for column 4 due to the structure of column 3. Furthermore the same remarks as above apply with regard to the moment we are allowed to branch.

Case 3: 2-∀∃-autarky $\{y_{i_1}, x_{i_2}\}$, with y_{i_1} ∀-quantified and x_{i_2} ∃-quantified, and $y_{i_1} < x_{i_2}$. We look at the preceeding example: In case $\mathfrak{I}(y_{i_1}) = 1$, then x_{i_2} becomes monotone, and only branching for column 1 is necessary. If $\mathfrak{I}(y_{i_1}) = 0$, also because of monotony only column 4 needs to be considered.

Case 4: 2-∃∀-autarky $\{x_{i_1}, y_{i_2}\}$, with x_{i_1} ∃-quantified, y_{i_2} ∀-quantified, and $x_{i_1} < y_{i_2}$. For a structure analogous to figure 9(b) on the previous page we only need to branch for columns 2 and 3 (similar to the 2-∀∀-autarky).

Figure 11: Two branchings for not-2-autarky.

All four cases have in common, that one reduction occurs because of the rule monotone literal. This may ease the later implementation. Also, not-autarky allows for simplifications as well, as shown in column 3 and 4 of figure 11.

VI. Changes to QBF Data Structures

From existing experiments with the rule monotone literal for SAT formulas it is known [7], that this rule does not lead to considerable improvements and is usually left out. For SAT solvers which use the described data structure the heuristic (and its computation cost) has considerable implications on the overall practical runtime of the solver. For QBF formulas the heuristic has less choice due to the prefix, on the other hand a wrong choice has stronger implications. Here we consider how to practically implement the proposed considerations by integrating them into our data structure.

For a quantified k-CNF clause α with n variables and m clauses the data structure requires $\mathcal{O}(k \cdot 2n + k \cdot m)$ space so far: each literal L has a field of length k that counts occurrences of L in clauses of length $1, \ldots, k$, where $k \cdot m$ is the size of the matrix. Therefore, the runtime of the lexicographic heuristic is $\mathcal{O}(k \cdot 2n)$. For QBF formulas the heuristic requires $\mathcal{O}(k \cdot 2 \cdot |Z_1|)$ time, where Z_1 denotes the leftmost prefix group.

In order to consider structural information for a given variable pair (z_1, z_2) for the heuristic `Choose-Literal()`, a field s of length 8 is used for each relevant combination, which counts the occurrence of the pairs (z_1, z_2) in the clauses of the formula. Only combinations of pairs are relevant that occur at least once together in a clause. For example: if x_1 and y_2 occur only as $(\ldots \vee x_1 \vee \ldots)$ or $(\ldots \vee \neg x_1 \vee \ldots)$ and $(\ldots \vee y_2 \vee \ldots)$ or $(\ldots \vee \neg y_2 \vee \ldots)$, the removal of a clause containing x_1 or $\neg x_1$ may not lead to y_2 becoming monotone. Therefore, structural information of at most $\mathcal{O}(k^2 \cdot m)$ many of $\mathcal{O}(n^2)$ possible pairs needs to be kept: for each of the m clauses only up to $\frac{k \cdot (k-1)}{2}$ new, so far not considered pairs may be introduced.

The fact that the set of $\mathcal{O}(k^2 \cdot m)$ many field addresses is static for a given input formula may be used for accelerated access: A supporting data structure must provide for fast access for any given variable pair. A possible solution for this is to use corresponding *hash functions*.

When using hashing, a set of keys $S \subseteq \mathcal{U}$ with universe $\mathcal{U} = \{1, \ldots, N\}$ and $|S| \ll |\mathcal{U}|$ is mapped to numbers $0, \ldots, t-1$ with $t \geq s := |S|$. Here, a hash function $h : \mathcal{U} \to \{0, \ldots, t-1\}$ is used. In case $h_{|S}$ is one-to-one, we call it a *perfect* hash function, which is per definitionem collision free. We cite from [10] the result: for every $t \geq 3 \cdot s$ there exists a perfekt hash function, which can deterministically be computed in time $\mathcal{O}(s \cdot N)$ and probabilistically in time $\mathcal{O}(s)$ and whose execution requires time $\mathcal{O}(1)$. One such hash function is described by a program with $\mathcal{O}(s \cdot \log N)$ bits. For a short overview on the subject we refer to [11].

Aside from hash functions we may also consider a hybrid data structure like the *trie* or *prefix trie* for our purposes. So [12] describes a variant of such a trie data structure suitable for storing a set $S \subseteq \mathcal{U}$, where $s := |S|$ and $N := |\mathcal{U}|$ are as above, with $\mathcal{O}(s)$ memory slots with $\mathcal{O}(\log s)$ bits each and worst-case access of $\mathcal{O}\big(\log_s(N)\big)$. For our case ($r$ in $r \cdot n = m$ denotes the ratio of clauses to variables) $N = n^2$

is polynomial in $s = k^2 \cdot m = k^2 \cdot r \cdot n$, which leads to the access time of $\mathcal{O}\big(\log_s(N)\big) = \mathcal{O}\big(\frac{\log N}{\log s}\big) = \mathcal{O}(1)$. The layout of the data structure requires some effort, therefore for first experiments a method is proposed that uses probabilistic methods to calculate an adequate hash function for the given instance.

The information on the structure of a given variable pair (z_1, z_2) may now be managed as follows: First for all relevant variable pairs corresponding memory is allocated (access time $\mathcal{O}(1)$) and the corresponding counters are initialized with 0; then, for each of the m clauses and therein for each of the $\mathcal{O}(k^2)$ pairs the corresponding 8 structure counters are updated. To later allow a clause $(L_1, \ldots, L_{\ell-1}, L_\ell)$ of length ℓ to be shortened by setting a literal to `false`, $\ell - 1$ counters must each be decreased by 1. In case a clause of length ℓ is removed, $\frac{\ell \cdot (\ell-1)}{2}$ many counters need to be decreased by 1. Accordingly, these operations have to be reverted in reverse order when returning from a lower recursion level. The relevant structural classes necessary to identify a 2-autarky as such may be deposited in tabular form, where the table can be generated during compile time.

Altogether, the space requirements for the data structure is increased from $\mathcal{O}(k \cdot 2n + k \cdot m)$ to $\mathcal{O}(k \cdot 2n + k^2 \cdot m)$. The time requirement to shorten a clause is still $\mathcal{O}(k)$, whereas the removal of a clause leads to the changed runtime requirement of $\mathcal{O}(k^2)$.

VII. Conclusion and future work

The strength of the described SAT data structure may also be observed for its extension to QBF: because of a small memory footprint along with its operation, formulas of considerable size fit into the CPU data cache, with small runtime requirements for elemenary operations. Up to date, recent QBF solvers in contrast to recent SAT solvers can only cope with comparatively small randomized instances of quantified boolean formulae [13], which shows the benefits of a compact data structure.

Therefore a paramatrized analysis of 2-autarky based SAT reductions seems promising to identify measures that significantly purge the QBF search tree. Also subject of future examinations is the analysis how to efficiently integrate and parametrize these SAT reductions with other implemented reductions (like trivial truth or trivial falsity), while still keeping the memory footprint of the corresponding QBF data structure small.

To the best of our knowledge, no research has been undertaken yet to utilize the detection of 2-autarky structures for pruning the search tree of existing QBF solvers.

VIII. Acknowledgments

The author would like to thank Ewald Speckenmeyer, Stefan Porschen, and Bert Randerath for many fruitful discussions and the reviewers who helped improve the original manuscript.

References

[1] G. Gopalakrishnan, Y. Yang, and H. Sivaraj, "QB or Not QB: An Efficient Execution Verification Tool for Memory Orderings," in *Proceedings of the 16th International Conference on Computer Aided Verification (CAV 2004)*, 2004, pp. 401–413.

[2] M. Mneimneh and K. Sakallah, "Computing Vertex Eccentricity in Exponentially Large Graphs: QBF Formulation and Solution," in *Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing (SAT 2003)*, 2003, pp. 411–425.

[3] J. Rühmkorf, "Entwicklung eines leistungsfähigen Lösers für Quantifizierte Boolesche Formeln," Master's thesis, University of Cologne, 2005.

[4] H. Kleine Büning and T. Lettman, *Propositional Logic: Deduction and Algorithms*. Cambridge University Press, 1999.

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

[6] M. Cadoli, A. Giovanardi, and M. Schaerf, "An Algorithm to Evaluate Quantified Boolean Formulae," in *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI 1998)*, Madison, WI, 26.–30. Jul. 1998, pp. 262–267.

[7] M. Böhm and E. Speckenmeyer, "A Fast Parallel SAT-Solver – Efficient Workload Balancing," *Annals of Mathematics and Artificial Intelligence*, vol. 17, pp. 381–400, 1996.

[8] R. Feldmann, B. Monien, and S. Schamberger, "A Distributed Algorithm to Evaluate Quantified Boolean Formulae," in *Proceedings of the 17th National Conference on Artificial Intelligence (AAAI 2000)*. Austin, TX: American Association of Artificial Intelligence, 30. Jul. – 3. Aug. 2000, pp. 285–290.

[9] B. Monien and E. Speckenmeyer, "Solving Satisfiability in less than 2^n Steps," *Discrete Applied Mathematics*, vol. 10, no. 3, pp. 287–295, Mar. 1985.

[10] M. L. Fredman, J. Komlós, and E. Szemerédi, "Storing a Sparse Table with O(1) Worst Case Access Time," *Journal of the ACM*, vol. 31, no. 3, pp. 538–544, Jul. 1984.

[11] K. Mehlhorn and A. K. Tsakalidis, "Data Structures," in *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity*, J. van Leeuwen, ed. Amsterdam: Elsevier, 1990, pp. 301–342.

[12] R. E. Tarjan and A. C.-C. Yao, "Storing a Sparse Table," *Communications of the ACM*, vol. 22, no. 11, pp. 606–611, Nov. 1979.

[13] "The Third Competitive Evaluation of QBF Solvers," 2008, http://www.qbflib.org/, last accessed 1. Jul. 2010.

 ISBN: 978-1-61208-101-4