Nenofex: Expanding NNF for QBF Solving

Florian Lonsing and Armin Biere

Johannes Kepler University

Abstract. The topic of this paper is Nenofex, a solver for quantified boolean formulae (QBF) in negation normal form (NNF), which relies on expansion as the core technique for eliminating variables. In contrast to eliminating existentially quantified variables by resolution on CNF, which causes the formula size to increase quadratically in the worst case, expansion on NNF is involved with only a linear increase of the formula size. This property motivates the use of NNF instead of CNF combined with expansion. In Nenofex, a formula in NNF is represented as a tree with structural restrictions in order to keep its size small and distances from nodes to the root short. Expansions of variables are scheduled based on estimated expansion cost. The variable with the smallest estimated cost is expanded first. In order to remove redundancy from the formula, limited versions of two approaches from the domain of circuit optimization have been integrated. Experimental results on latest benchmarks show that Nenofex indeed exceeds a given memory limit less frequently than a resolution-based QBF solver for CNF, but also that there is the need for runtime-related improvements.

1 Introduction

QBF [36, 19] is an important research area with many applications [28, 33, 6, 31, 21, 16, 24, 5, 30]. Progress has been impressive in recent years, but in practice QBF solvers lack the generic applicability on PSPACE hard problems as SAT solvers on NP hard problems.

We believe that one of the reasons is the usage of CNF as input *and* as internal format to reason about SAT problems. We argue that in QBF more general data structures are necessary. There is clear indication in the relevant literature [39, 37, 34] that supports this conjecture.

The most natural extension of CNF is NNF: NNF is still tree shaped, i.e. there is no sharing, and a formula in CNF is as well a formula in NNF. Before mentioned approaches [39, 34] and in essence all QBF solvers that learn solutions [29, 40, 20] use some kind of combination of DNF and CNF, which again can be considered NNF. As we argue in this paper, a tree representation has many advantages compared to a more general DAG representation.

We investigated the usage of NNF to reduce space usage of expansion based QBF solvers. Our experiments clearly indicate, that our prototype implementation Nenofex needs less space than the highly optimized expansion based solver Quantor [7], which works on CNF. Nenofex is also able to solve several instances, that could not be handled by Quantor.

The success of Quantor is based on two techniques. First, it implements a fast scheduling algorithm for heuristically choosing the next variable to expand. The second contribution is a fast subsumption algorithm, that removes redundant clauses generated during expansions. Both techniques are also crucial for the efficiency of the SAT preprocessor SATeLite [17]. To maintain accurate expansion costs for NNF turns out to be much more difficult than for CNF. The same comment applies to redundancy removal for NNF. In Nenofex we considered both problems, but due to space constraints only present a solution to redundancy removal on NNF in this paper.

Another related application of a restricted form of NNF is knowledge compilation [12]. But it is unclear how to use it for QBF solving.

2 Motivation

In order to show that NNF is much more compact than CNF for representing the *result* of expansion, consider the following formula $F \equiv R \wedge X_0 \wedge X_1$ in CNF. The clause sets $X_0 = \{c_1, c_2, c_3\} = \{(\neg x \vee c \vee \neg d), (\neg x \vee d \vee \neg e), (\neg x \vee e \vee \neg c)\}$ and $X_1 = \{c_4, c_5, c_6\} = \{(x \vee f \vee \neg g), (x \vee g \vee \neg h), (x \vee h \vee \neg f)\}$ with $|X_0| = |X_1| = 3$ contain all clauses with negative and positive literals of variable x, respectively. Variable x has $n = 3$ negative and $p = 3$ positive literals. $R = \{(a \vee b)\}$ is the set of clauses which do not contain a literal of x (notation adopted from [14]).

Variable x may be expanded by copying F: $F \equiv F[x/0] \vee F[x/1]$ where expression $F[x/v]$ denotes the formula obtained from F by substituting value v for every literal of x. This yields

$$F \equiv ((R \wedge X_0 \wedge X_1)[x/0]) \vee ((R \wedge X_0 \wedge X_1)[x/1]) \quad (1)$$

$$F \equiv (R \wedge (X_0 \wedge X_1)[x/0]) \vee (R \wedge (X_0 \wedge X_1)[x/1]) \quad (2)$$

$$F \equiv R \wedge ((X_0 \wedge X_1)[x/0] \vee (X_0 \wedge X_1)[x/1]) \quad (3)$$

$$F \equiv R \wedge (X_0' \vee X_1') \quad (4)$$

which is $(a \vee b) \wedge (((c \vee \neg d) \wedge (d \vee \neg e) \wedge (e \vee \neg c)) \vee ((f \vee \neg g) \wedge (g \vee \neg h) \wedge (h \vee \neg f)))$. In the clause set X_0' (X_1') all negative (positive) literals of variable x have been deleted. Clauses in R have not been affected during expansion, hence this set can be factored out as shown in formula 3. Formulae 1 to 4 are not in CNF any more but in NNF.

If x is eliminated by resolution, then the set of resolvents $X_r = \{c_{i,j} \mid i = 1, \ldots, n, j = n+1, \ldots, n+p,\ c_{i,j} = (c_i \cup c_j) \setminus \{x, \neg x\}\}$ contains clauses $\{c_{1,4}, c_{1,5}, c_{1,6}, c_{2,4}, c_{2,5}, c_{2,6}, c_{3,4}, c_{3,5}, c_{3,6}\}$ where $|X_r| = n \cdot p = 3 \cdot 3 = 9$ clauses. After discarding sets X_0 and X_1 and adding X_r to F, we have $F \equiv (a \vee b) \wedge (c \vee \neg d \vee f \vee \neg g) \wedge (c \vee \neg d \vee g \vee \neg h) \wedge (c \vee \neg d \vee h \vee \neg f) \wedge (d \vee \neg e \vee f \vee \neg g) \wedge (d \vee \neg e \vee g \vee \neg h) \wedge (d \vee \neg e \vee h \vee \neg f) \wedge (e \vee \neg c \vee f \vee \neg g) \wedge (e \vee \neg c \vee g \vee \neg h) \wedge (e \vee \neg c \vee h \vee \neg f)$.

Expanding a variable in a formula can at most double its size. In particular, expanding some variable x on CNF as shown in formulae 1 to 4 will copy $n + p$ clauses (and introduce one logical connective), whereas eliminating the same variable by resolution will produce $n \cdot p$ clauses in the worst case. Expansion

causes a formula to grow linearly in contrast to resolution, which is involved with a quadratic size increase. In the example, this is reflected in the sizes of the formulae resulting from expansion and resolution.

It is exactly this observation which motivates the use of NNF as formula representation in an expansion based QBF solver as Nenofex. We expect less size increase when eliminating existential variables by expansion on NNF than by resolution on CNF. When expanding universal variables, there is no advantage of expansion on NNF compared to CNF because in both cases the same set of clauses has to be copied. For an arbitrary formula in NNF, expansion of a variable will always yield a formula which is in NNF again. No transformations need to be carried out on the formula between expansions.

3 Preliminaries

For a set of variables V, a *literal* (or *occurrence*) is either a variable $v \in V$ or its negation $\neg v$. A *clause* is a disjunction over literals. A (propositional) formula is in *conjunctive normal form (CNF)* if it is a conjunction over clauses. A formula consisting of disjunctions, conjunctions and literals is in *negation normal form (NNF)* if the negation operator is applied to literals only.

A *quantified boolean formula (QBF)* $F \equiv S_1 S_2 \ldots S_n \, \phi$ consists of a propositional formula ϕ over a set of variables V and a *quantifier prefix* $S_1 S_2 \ldots S_n$. We assume that ϕ is in NNF. The quantifier prefix is an ordered set of *scopes* S_i, which forms a partition on the set of variables: $V = S_1 \cup S_2 \cup \ldots \cup S_n$ and $S_i \cap S_j = \emptyset$ for $1 \leq i, j \leq n$ and $i \neq j$. A scope S_i is *existential* (*universal*) if it is associated with an existential (universal) quantifier, written as $\mathit{type}(S_i) = \exists$ ($\mathit{type}(S_i) = \forall$). A variable $x \in S_i$ where $\mathit{type}(S_i) = \exists$ ($\mathit{type}(S_i) = \forall$) is existentially (universally) quantified. By convention, for two adjacent scopes S_i and S_{i+1}, where $1 \leq i < n$, $\mathit{type}(S_i) \neq \mathit{type}(S_{i+1})$.

The set of scopes is a linearly ordered set $S_1 < S_2 < \ldots < S_n$ which follows from the order of appearance of scopes S_i in the quantifier prefix. Scope S_1 is the outermost, scope S_n the innermost scope. Variables are ordered with respect to the order of scopes they belong to. For variables from the same scope, an arbitrary order may be chosen. Our definitions of QBF and scopes are similar to the ones in [7] except that formula ϕ is not in CNF but NNF.

There is strong indication that non-prefix orders are important for QBF reasoning [18, 7, 3, 22]. Initially we experimented with non-prefix orders as well, but due to the complexity involved, we focus on non-CNF representation in this paper, except for on-the-fly miniscoping during expansions as in Quantor.

A *tree* $T = (N, E)$ consists of a set of nodes N and a set of directed edges $E \subseteq N \times N$ such that for exactly one node r called *root*, there is no $v \in N$ where $(v, r) \in E$, and for each node $w \in N \setminus \{r\}$, there exists exactly one $v \in N$, $v \neq w$, such that $(v, w) \in E$. If $(v, w) \in E$, then v is the *parent* of w and w is a *child* of v. The root is the only node in T which has no parent, any other node has exactly one parent. For nodes v and $w \in N$, a *path* of length k from v to w is a sequence of nodes $p_0, p_1, \ldots, p_k$ where $p_0 = v$, $p_k = w$ and $(p_i, p_{i+1}) \in E$ for

$0 \leq i < k$. For any node v, there is a path of length 0 from v to v. The *level* of a node v is the length of the path from the root to v. For root r, $level(r) = 0$. If $(v, w) \in E$, then $level(w) = level(v) + 1$. For nodes v and w, if there is a path from v to w, then v is an *ancestor* of w and w is a *descendant* of v. Every node is ancestor and descendant of itself. The root is ancestor of any node in T.

A *common ancestor* of a pair of nodes v,w in T is a node which is ancestor of both v and w. The *least common ancestor (LCA)* of v,w, written as $lca(v, w)$ where $lca : N \times N \rightarrow N$, is their common ancestor with maximum level, that is, which is farthest away from the root. Commutativity and associativity of *lca* as an operator extend the definition from pairs to sets of nodes:

$$lca(n_1, n_2, \ldots, n_k) = \begin{cases} lca(lca(n_1, n_2), n_3, \ldots, n_k) & if \quad k \geq 3 \\ \text{least common ancestor of } n_1 \text{ and } n_2 & if \quad k = 2 \\ n_1 & if \quad k = 1 \end{cases}$$

4 Formula Representation

A formula ϕ in NNF is represented as a tree $T = (N, E)$, referred to as *NNF-tree*. The set of nodes N is partitioned into *operator nodes* N_O and *literal occurrence nodes* N_L (short: *literal nodes*), that is $N = N_O \cup N_L$ and $N_O \cap N_L = \emptyset$. The set N_O (N_L) comprises exactly the set of internal nodes (leaf nodes) of the tree.

The set N_O is partitioned into the sets $N_\vee$ and $N_\wedge$, that is $N_O = N_\vee \cup N_\wedge$ and $N_\vee \cap N_\wedge = \emptyset$. A node from the set $N_\vee$ ($N_\wedge$) is called *OR-node* (*AND-node*) and denotes the logical disjunction (conjunction) over its children. Operator nodes with n children, where $n \geq 2$, represent n-ary boolean functions.

A node $n_l \in N_L$ denotes one single (positive or negative) literal of some variable $x \in V$. Conversely, a literal of some variable x is represented by exactly one node $n_l \in N_L$. The *least common ancestor (LCA) of a variable* $x \in V$, written as $lca(x)$, is defined as the LCA over all of its literal nodes.

The structure of an NNF-tree is restricted as follows. Operator nodes may have an arbitrary number of children but must have at least two. For operator nodes $n_o \in N_\vee$ ($n_o \in N_\wedge$) and all its children c, either $c \in N_\wedge$ or $c \in N_L$ ($c \in N_\vee$ or $c \in N_L$), that is, the types of operator nodes and their children must be different. This corresponds to the application of associativity of disjunction and conjunction whenever possible. For operator nodes $n_o \in N_O$ and some variable $x \in V$, if n_o has a child $c_1 \in N_L$ which is a literal node of x, then n_o must not have another child $c_2 \in N_L$, $c_1 \neq c_2$, which is a literal node of x. Thus operator nodes must neither have multiple nor complementary literals of one and the same variable as children. The structural restrictions aim at keeping the NNF-tree small and node levels, that is distances between nodes and the root short. Fig. 1 shows a sample NNF-tree.

As an alternative to trees, a representation related to *directed acyclic graphs (DAGs)* could have been used, which allow nodes to be *structurally shared* among several parents. A well-known, DAG-related formula representation are *And-Inverter Graphs (AIGs)* [26] where the set of operators is restricted to binary conjunction and negation. Methods for identifying structural sharing in AIGs

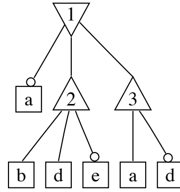

Fig. 1. NNF-tree for formula $\neg a \vee (b \wedge d \wedge \neg e) \vee (a \wedge \neg d)$. An AND-node (OR-node) is represented as a triangle △ (inverted triangle ▽) resembling the symbol for conjunction $\wedge$ (disjunction $\vee$), and a literal node as a box □. A circle ∘ at a literal node denotes the negation operator. Labels of operator nodes in the figures are used as indices in the text, e.g. n_1 denotes the root of the NNF-tree in the example above.

have been studied in [9, 8]. To our knowledge, structural sharing in combination with n-ary operators like in an NNF-tree has not been studied at a large extent, but obviously there is much more complexity involved. Furthermore, NNF-trees guarantee that a formula in CNF has a flat representation: each (non-unit) literal in the CNF corresponds to exactly one literal node with level 2, each clause to exactly one OR-node with level 1 and the conjunction over the clauses to one single AND-node at the root of the tree. It is impossible to achieve these correspondences with AIGs. DAGs complicate the implementation. For each node, the set of parents and children need to maintained under modifications of the graph. With trees, algorithms related to expansion (next section) and redundancy removal (section 6.1) are much easier to implement.

5 NNF Expansion

If a variable is expanded as shown in the introductory example (formulae 1 to 4), then parts of the formula might be copied unnecessarily and need to be factored out in order to reduce the size of the expanded formula.

We present *local expansion* for NNF, a method where only the relevant parts of a formula are copied and which does not require factoring out common subformulae in the expanded formula. Generally, our method can be regarded as *miniscoping* [2], which produces a non-prefix reduced scope through the application of standard quantifier rules, followed by expansion. In our approach (section 5.3), a minimal reduced scope is determined bottom up, starting from the literal occurrences of the expanded variable.

For a QBF $S_1 \dots S_{n-1} S_n \, \phi$, we consider expansion of (1) existential or (2) universal variables from scope S_n (section 5.1), and expansion of (3) universal variables from scope S_{n-1} (section 5.2) only. Case (2) is irrelevant for formulae in CNF since *forall-reduction* [25, 7] could remove all literals of universal variables in S_n instead: a universal literal can be removed from a clause if there is no existential literal in that clause whose variable belongs to a scope which is larger than the scope of the universal literal's variable. To our knowledge, it is not clear

whether and how this operation can be applied to formulae in NNF. Replacing innermost universal literals by *false* is incorrect, because this would reduce the following formulae to *false* even though they are valid:

$$\forall x\ (x \vee \overline{x}) \qquad\qquad \forall x, y\ (xy \vee x\overline{y} \vee \overline{x}y \vee \overline{xy})$$

5.1 Innermost Expansion

Given a QBF $S_1 \dots S_n\ \phi$ and some variable x in S_n where $type(S_n) = \exists$ or $type(S_n) = \forall$, let $ers(x)$ denote the *expansion relevant subformula* of variable x, which is the smallest subformula of ϕ which contains all literals of x.

Local expansion of variable x in ϕ is defined as follows:

$$S_1 \dots S_n\ \phi \equiv S_1 \dots (S_n \setminus \{x\})\ \phi[ers(x)\ /\ (ers(x)[x/0] \otimes ers(x)[x/1])] \qquad (5)$$

where operator $\otimes = \vee$ ($\otimes = \wedge$) if $type(S_n) = \exists$ ($type(S_n) = \forall$). In rule 5, ϕ is modified by replacing the expansion relevant subformula $ers(x)$ by a subformula consisting of two copies of $ers(x)$, where variable x is assigned *true* resp. *false*.

5.2 Non-innermost Expansion

Expansion of universal variables from scope S_{n-1} requires depending existential variables from S_n to be duplicated. Concerning CNF, methods for universal expansion and for identifying dependencies have been proposed in Quantor [7], sKizzo [4], quantifier trees [3] and bounded universal expansion [10]. For example, before some universal variable x from scope S_{n-1} is expanded in Quantor, the set of depending existential variables from scope S_n is computed via a connection relation. Then, all clauses which contain literals of x or of any depending variable are copied during expansion. This idea is generalized in [10, 35].

Given a QBF $S_1 \dots S_{n-1} S_n\ \phi$ with n scopes and some universal variable x in S_{n-1} where $type(S_{n-1}) = \forall$ and $type(S_n) = \exists$. Let $ers(x)$ be defined as in the previous section. Let D_x be the set of *depending existential variables* of x defined as follows (notation adopted from [10]):

$$\begin{aligned} D_x^{(0)} &:= \{y \in S_n \mid y \text{ has literals in } ers(x)\} \\ D_x^{(k+1)} &:= \{z \in S_n \mid z \text{ has literals in } ers(y') \text{ for some } y' \in D_x^k\}, k \geq 0 \\ D_x &:= \bigcup_k D_x^k \end{aligned}$$

Let $urs(x, D_x)$ denote the *expansion relevant subformula* of universal variable x with respect to D_x, which is the smallest subformula of ϕ which contains all literals of x and all literals of any existential variable $y \in D_x$. *Local expansion* of variable x in ϕ is defined as follows:

$$\begin{aligned} &S_1 \dots S_{n-1} S_n\ \phi \equiv \\ &\quad S_1 \dots (S_{n-1} \setminus \{x\})(S_n \cup D'_x)\ \phi[u\ /\ (u[x/0] \wedge u'[x/1])] \end{aligned} \qquad (6)$$

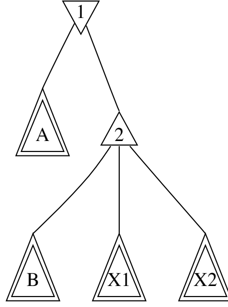

Fig. 2. NNF-tree for formula $A \vee (B \wedge X_1 \wedge X_2)$.

where u stands for $urs(x, D_x)$ and $urs(x, D_x)'$ is obtained from $urs(x, D_x)$ by substituting y' for all literals of $y \in D_x$. D'_x is the set which contains duplicated variables y' for every $y \in D_x$. The definition of *urs* extends the one of *ers* from the previous section by taking the set of depending existential variables into account. In fact, the notion of $urs(x, D_x)$ is closely related to the CNF-based approaches in [7] and [10], where the set D_x is constructed via a connection relation between variables: v_i is locally connected to v_j if both occur in a common clause. In our NNF-based approach, the connection relation is generalized to subformulae.

5.3 Expansion Relevant LCAs

According to the definitions of expansion relevant subformulae *ers* and *urs* for some variable x in some formula ϕ, the *expansion relevant subtree* of x is defined to be the smallest subtree in the NNF-tree of ϕ which contains all literals of x.

In order to expand x in the NNF-tree, a correspondence between expansion relevant subformulae and subtrees as defined has to be established. The *expansion relevant LCA* of variable x is defined by node $lca(x)$ and the set of *LCA-children* of $lca(x)$. A child of node $lca(x)$ is an LCA-child if its subtree contains at least one literal of x. The subtree denoted by the expansion relevant LCA exactly corresponds to the expansion relevant subformula and vice versa.

In Fig. 2, subtrees X_1 and X_2 contain all literals of some variable x and node n_2 is $lca(x)$. The roots of subtrees X_1 and X_2 form the set of LCA-children and, together with node $lca(x)$, denote the expansion relevant subtree of x, which corresponds to $X_1 \wedge X_2$, the expansion relevant subformula of x. Generally, LCAs of variables without the notion of LCA-children are only an over-approximation for expansion relevant subtrees. In Fig. 2, the subtree of node $lca(x)$ contains subtree B as well, which does not contain literals of x.

In Nenofex, expansion relevant LCAs are computed incrementally in an upward directed search starting from each literal of the variable, where parents are successively visited and LCA-children are collected. Our approach requires $O(nm)$ time in the worst case, where n is the number of literals and m the maximum level of a literal which is expected to be small due to structural restrictions.

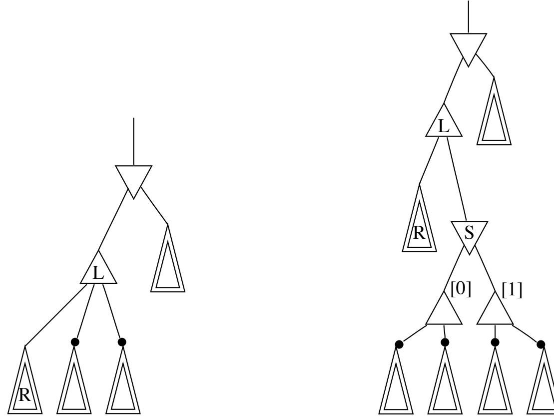

Fig. 3. Expansion template. Node n_L is $lca(x)$ for some existential variable x, subtree R does not contain literals of x and black dots indicate LCA-children. In the right NNF-tree, x has been expanded: OR-node n_S is parent of two copies of the expansion relevant subtree, where x is assigned *true* ([1]) and *false* ([0]).

In order to expand a variable in an NNF-tree, its expansion relevant subtree needs to be copied. Fig. 3 illustrates the situation for an existential variable whose LCA is an AND-node and which has two LCA-children. Expanding variable x in the formula in Fig. 2 yields the expanded formula $A \vee (B \wedge ((X_1 \wedge X_2)[x/0] \vee (X_1 \wedge X_2)[x/1]))$, as indicated by the right template in Fig. 3.

6 Implementation

The architecture of Nenofex is very similar to the one of Quantor. Variables are eliminated in cyclic fashion starting from the innermost scope, where scheduling is based on estimated elimination costs. Elimination of variables is interspersed with redundancy removal. If there is only one type of variables left, then the QBF is reduced to a SAT problem and forwarded to an internal SAT solver.

Fig. 4 shows the phases of the core algorithm in Nenofex. In either phase, the solver may return an answer if the NNF-tree has been deleted or the SAT solver has terminated. After an initialization phase (INIT in Fig. 4), where the problem instance is parsed and data structures are set up, *unit literals* and *pure literals* (or *unates*) [11] are eliminated until saturation.

6.1 Redundancy Removal

Local expansion avoids unnecessary copies of formula parts, but can not avoid redundancy in general. As in Quantor, which relies on subsumption checking, redundancy removal is crucial for Nenofex to achieve best performance. For this purpose, limited versions of two approaches from the domain of circuit optimization have been implemented where an NNF-tree is regarded as a circuit.

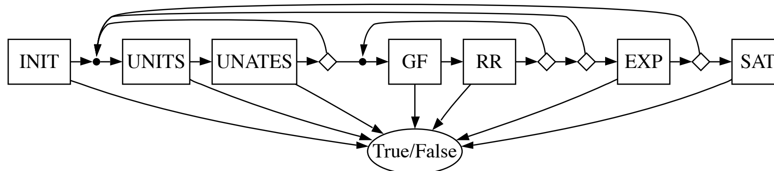

Fig. 4. Core algorithm of Nenofex. Parsing and initialization (INIT), elimination of units and unates (UNITS, UNATES), global flow (GF), redundancy removal (RR), expansion (EXP) and propositional SAT solving (SAT).

ATPG-based Redundancy Removal In *structural testing*, a test for a fault at a single line or gate in a circuit is a set of input values, called *test pattern*, by which wrong circuit outputs related to the faulty part can be detected. Test patterns can be generated algorithmically, which is the main purpose of *automatic test pattern generation (ATPG)* [1]. A fault which does not change the circuit's behaviour is redundant and the respective hardware may be removed.

A typical model for faults related to single lines (or signals) is the *stuck-at-fault model.* A line is stuck-at-1 (stuck-at-0), if it always carries *true* (*false*) regardless of the intended value. Detection and removal of redundant stuck-at faults can be used for circuit optimization. Testing a stuck-at fault in *ATPG-based redundancy removal* [1] comprises three steps. In *fault sensitization*, the fault is activated by assigning the corresponding signal the opposite value of the fault: for a stuck-at-1 fault, the signal is assigned *false*, otherwise *true.*

In *path sensitization*, the effect of the activated fault must be propagated *unambiguously* along a *fault path* to an output signal of the circuit. This can be achieved by assigning conservative values to all off-path inputs of gates along the fault path. Off-path inputs of OR-gates (AND-gates) must be assigned *false* (*true*). There might be exponentially many fault paths. If propagation on one path fails, then possibly all remaining paths have to be considered.

In *justification*, signal assignments made in the previous two steps must be justified by finding a set of circuit inputs which establish the configuration of internal signal assignments. Starting at an unjustified, assigned signal, its inputs are assigned recursively with justifying values. For example, an AND-gate which is assigned *false* may be justified by assigning *false* to one of its inputs. As in DPLL-based SAT solvers [13], justification involves making *decisions* which have to be undone during *backtracking* if conflicts between assignments occur.

If all fault paths and alternative assignments have been tried out but conflicts could not be resolved, then the fault is untestable: there is no set of input values such that the fault effect can be observed unambiguously at a circuit output. The corresponding hardware is redundant and may be removed, which can cause further faults to become redundant.

Global Flow *Global flow* [27] is an approach for circuit minimization where implications are derived from signals which are then used to transform the circuit

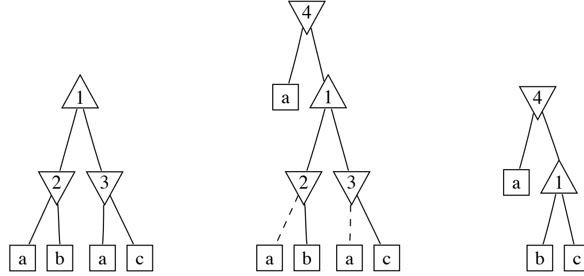

Fig. 5. Detecting distributivity by global flow and redundancy removal.

without changing the logical flow of values. For any signal x in the circuit, there are four sets of implications defined as $F_{VW}(x) := \{s : x = V \rightarrow s = W\}$ where $V, W \in \{0, 1\}$ and s is a signal. Given the sets F_{VW} for some signal x, the following transformations are valid:

$$y \in F_{00}(x) : y \equiv x \wedge y \qquad y \in F_{10}(x) : y \equiv \neg x \wedge y$$
$$y \in F_{11}(x) : y \equiv x \vee y \qquad y \in F_{01}(x) : y \equiv \neg x \vee y$$

In order to optimize a circuit, first some signal x is chosen where subsets of implications in $F_{VW}(x)$ are computed, because full computation is complex. Next, an implication is chosen and the circuit is transformed according to the respective rule. Certain connections of x to other nodes may be removed, provided that the logical flow of the value from x to the implied node does not change. If circuit size is not decreased, then all modifications will be are reversed. These steps are carried out in cyclic fashion for all signals in the circuit.

Fig. 5 illustrates a typical situation where redundancy is detected by global flow *together with* ATPG-based redundancy removal: in the leftmost NNF-tree, literal a may be factored out by applying distributivity. This can not be detected by ATPG-based redundancy removal alone. When assigning literal a at n_2 *true* (and consequently variable a as well) in the leftmost NNF-tree, then n_1 (the root) will be assigned *true* as well, hence $a = \mathit{true} \rightarrow n_1 = \mathit{true}$ and n_1 may be replaced by $a \vee n_1$ which yields the second NNF-tree. Dashed edges indicate nodes with untestable stuck-at-0 faults. If the fault at literal a at n_2 is tested, then it must be assigned *true* in fault sensitization (that is, variable a will be assigned *true*), but this yields an unresolvable conflict at n_4, the only circuit output, where literal a will be assigned *true* as well. This is not a conservative value as required by path sensitization. The same argument applies for a at n_3, and the two literals may be removed which yields the NNF-tree on the right.

Limitations The implementation of global flow (phase GF in Fig. 4) and ATPG-based redundancy removal (phase RR) is very limited. For GF, only implications from sets F_{00} and F_{11} are considered. GF alone is not capable of reducing the size of an NNF-tree but, together with RR (as shown in Fig. 5), can

enable detection of redundancies which would remain undetected by RR. Modifications made by GF are never reversed, since they always produce additional redundant stuck-at-faults due to the tree shape of NNF.

General ATPG-based redundancy removal is NP-complete [1]. Our implementation runs in polynomial time, but is incomplete. We only use propagations and no decisions. It greatly benefits from the tree representation of NNF, because there is a single fault propagation path from the fault site to the root.

Phases GF and RR are carried out in cyclic fashion on a small subtree of the NNF-tree only. Generally, each optimization runs until saturation, but this can become problematic due to the amount of required runtime. Therefore, fixed limits are imposed on the size of the subtree and on the number of value propagations during GF and RR.

6.2 Expansion

Let S_i and S_{i-1} denote the current innermost and first non-innermost scope, respectively. Variables are selected for expansion depending on their estimated expansion costs (scores) and on the types of S_i and S_{i-1}. In Nenofex, generally a greedy strategy is applied: in order to keep the size of the NNF-tree small in each expansion, always the variable with minimum expansion cost is selected.

First, if $\textit{type}(S_{i-1}) = \exists$ and $\textit{type}(S_i) = \forall$, then only variables from S_i may be expanded. The variable with minimum score is expanded. Second, if $\textit{type}(S_{i-1}) = \forall$ and $\textit{type}(S_i) = \exists$, then variables from both scopes may be expanded. A variable from S_{i-1} is eligible for expansion iff the *preceding* expansion from S_i (1) caused the size of the NNF-tree to increase and (2) the size increase to exceed a heuristic *universal threshold* t_u. Initially, t_u is set to 10 nodes. If t_u is exceeded in an expansion, then expansion of *exactly one* variable from S_{i-1} will be *forced*, which causes t_u to increase by 10 and expansions from S_{i-1} to be disabled again. Expansions are forced because score computation for S_{i-1} likely becomes impractical when carried out each time before an expansion. This policy goes against the greedy expansion strategy because the minimum score variable from S_i may well be cheaper than the one in S_{i-1}.

The estimated expansion cost is calculated as a tight upper bound on the real expansion cost. It is measured in the number of nodes added to and removed from the NNF-tree. The costs are recalculated after every change to the NNF-tree, taking only the changed part into account. The details are complicated and due to space constraints have to be omitted. However, we clearly see a potential for speeding up this process by updating scores in a full incremental fashion.

6.3 SAT Solving

If the formula contains only one type of variables, then the QBF may be reduced to a SAT problem: a formula in CNF is generated from the NNF-tree which is forwarded to an internal SAT solver. If only existential variables are left, then a

	Quantor	Nenofex		
		GF, RR	*no GF, RR*	*no GF, no RR*
Solved	**421**	361	352	313
OOT	**32**	124	103	83
OOM	683	**651**	681	740
MEM$\cup$	1.10e6	1.15e6	1.17e6	1.23e6
MEM$\cap$	10473	18472	19693	28422

Table 1. Comparison of Quantor and Nenofex in three different versions by number of instances where solvers succeeded, timed out (OOT) and ran out of memory (OOM). The number of instances solved by Nenofex decreases from (GF, RR) to (no GF, no RR), which indicates that GF and RR contribute positively to the solver's performance. On the other hand, time is traded for memory when enabling optimizations: values of OOT increase from (no GF, no RR) to (GF, RR), the opposite effect can be observed for memory. Note that (no GF, no RR) runs out of memory more often than Quantor, which applies subsumption checking, hence optimizations are crucial in combination with NNF as well. The last two lines show sums of actual maximum amounts of memory (in MB) consumed on each solved or unsolved instance (*MEM*$\cup$) and on instances solved by all four solvers (*MEM*$\cap$). The (unoptimized) node representation in Nenofex requires about twice as much memory than the pointer-based structures of clauses and literals in Quantor, which is reflected in the last line.

CNF will be generated which is satisfiable iff the formula denoted by the NNF-tree is *satisfiable*. Otherwise, a CNF will be generated which is satisfiable iff the formula denoted by the NNF-tree is *not a tautology*.

The algorithm for generating a CNF from an NNF-tree requires linear time in the number of nodes of the tree and is based on the Tseitin transformation [38]. Ideas from [15, 32] are combined to reduce the number of clauses in the resulting CNF.

7 Experiments

Nenofex was compared to Quantor on the benchmark collection used for the competitive QBF evaluation in 2007 [23], which contains 1136 instances. Both solvers used the same version of PicoSAT as backend SAT solver. Tests were run on a cluster of Pentium IV 3 GHz workstations running Linux, where runtime and memory were limited by 900 seconds and 1.5 GB, respectively.

Concerning global flow (GF) and redundancy removal (RR), Nenofex was run in three versions: either both GF and RR are enabled (GF, RR), or only RR is enabled (no GF, RR) or both GF and RR are disabled (no GF, no RR). In either version, the size of the subtree subject to these optimizations was limited by 500 nodes. Table 1 shows an overall comparison of Quantor and Nenofex, and in Tab. 2, behaviour unique to each solver is indicated.

	Quantor only	**Both**	**Nenofex only**	**Sum**
Solved	79	342	19	440
OOT	18	14	110	142
OOM	80	603	48	731

Table 2. Number of instances where both or only one of Quantor and Nenofex (GF, RR) succeeded, timed out or ran out of memory. Nenofex solved 19 instances which Quantor could not solve. OOT and OOM indicate a similar tendency as Tab. 1.

8 Conclusion

This paper showed that expansion on quantified NNF needs less space than CNF. However, it may be worthwhile to extend our algorithms to DAGs. So far we have only used NNF in an expansion based approach, which eliminates quantifiers from inside to the outside. As future work, one should also consider the combination of NNF with DPLL style algorithms. Finally we want to thank Ofer Strichman for very fruitful discussions on this subject.

References

1. V. Agrawal and M. Bushnell. *Essentials of Electronic Testing for Digital, Memory and Mixed-Signal VLSI Circuits.* Kluwer, 2000.
2. A. Ayari and D. A. Basin. QUBOS: Deciding quantified boolean logic using propositional satisfiability solvers. In *Proc. FMCAD'02*.
3. M. Benedetti. Quantifier trees for QBFs. In *Proc. SAT'05*.
4. M. Benedetti. sKizzo: A suite to evaluate and certify QBFs. In *Proc. CADE'05*.
5. Marco Benedetti. Experimenting with QBF-based formal verification. In *Proc. CFV'05*.
6. A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Proc. TACAS'99*.
7. Armin Biere. Resolve and expand. In *Proc. SAT'04*.
8. P. Bjesse and A. Borälv. DAG-aware circuit compression for formal verification. In *Proc. ICCAD'04*.
9. R. Brummayer and A. Biere. Local two-level and-inverter graph minimization without blowup. *Proc. MEMICS'06*.
10. U. Bubeck and H. Kleine Büning. Bounded universal expansion for preprocessing QBF. In *Proc. SAT'07*.
11. M. Cadoli, A. Giovanardi, and M. Schaerf. An algorithm to evaluate quantified boolean formulae. In *Proc. AAAI/IAAI'98*.
12. A. Darwiche. Decomposable negation normal form. *JACM*, 48(4), 2001.
13. M. Davis, G. Logemann, and D. W. Loveland. A machine program for theorem-proving. *CACM*, 5(7), 1962.
14. M. Davis and H. Putnam. A computing procedure for quantification theory. *JACM*, 7(3), 1960.
15. T. Boy de la Tour. An optimality result for clause form translation. *Symb. Comput.*, 14(4), 1992.

16. N. Dershowitz, Z. Hanna, and J. Katz. Bounded model checking with QBF. In *Proc. SAT'05*.
17. Niklas Eén and Armin Biere. Effective preprocessing in SAT through variable and clause elimination. In *Proc. SAT'05*.
18. U. Egly, M. Seidl, H. Tompits, S. Woltran, and M. Zolda. Comparing different prenexing strategies for quantified boolean formulas. In *Proc. SAT'03*.
19. M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* 1979.
20. E. Giunchiglia, M. Narizzano, and A. Tacchella. Learning for quantified boolean logic satisfiability. In *Proc. AAAI'02*.
21. E. Giunchiglia, M. Narizzano, and A. Tacchella. QBF reasoning on real–world instances. In *Proc. SAT'04*.
22. E. Giunchiglia, M. Narizzano, and A. Tacchella. Quantifier structure in search based procedures for QBFs. In *Proc. DATE'06*.
23. E. Giunchiglia, M. Narizzano, and A. Tacchella. Quantified Boolean Formulas satisfiability library (QBFLIB), 2001. www.qbflib.org.
24. T. Jussila and A. Biere. Compressing BMC encodings with QBF. In *Proc. BMC'06*.
25. H. Kleine Büning, M. Karpinski, and A. Flügel. Resolution for quantified boolean formulas. *Inf. Comput.*, 117(1), 1995.
26. A. Kuehlmann, V. Paruthi, F. Krohm, and M. K. Ganai. Robust boolean reasoning for equivalence checking and functional property verification. *TCAD*, 21(12), 2002.
27. W. Kunz and D. Stoffel. *Reasoning in Boolean Networks: Logic Synthesis and Verification Using Testing Techniques.* Kluwer, 1997.
28. R. Ladner. The computational complexity of provability in systems of modal propositional logic. *SIAM Journal on Computing*, 6(3), 1977.
29. R. Letz. Lemma and model caching in decision procedures for quantified boolean formulas. In *Proc. TABLEAUX'02*.
30. H. Mangassarian, A. Veneris, S. Safarpour, M. Benedetti, and D. Smith. A performance-driven QBF-based iterative logic array representation with applications to verification, debug and test. In *Proc. ICCAD'07*.
31. C. Otwell, A. Remshagen, and K. Truemper. An effective QBF solver for planning problems. In *MSV/AMCS*, 2004.
32. D. Plaisted and S. Greenbaum. A structure-preserving clause form translation. *Symb. Comput.*, 2(3), 1986.
33. J. Rintanen. Constructing conditional plans by a theorem-prover. *Journal of Artificial Intelligence Research*, 10, 1999.
34. A. Sabharwal, C. Ansótegui, C. Gomes, J. Hart, and B. Selman. QBF modeling: Exploiting player symmetry for simplicity and efficiency. In *Proc. SAT'06*.
35. M. Samer and S. Szeider. Backdoor sets of quantified boolean formulas. In *Proc. SAT'07*.
36. L. Stockmeyer. The polynomial–time hierarchy. *TCS*, 3(1), 1976.
37. D. Tang and S. Malik. Solving quantified boolean formulas with circuit observability don't cares. In *Proc. SAT'06*.
38. G. Tseitin. On the complexity of derivation in propositional calculus. *Studies in Constructive Mathematics and Mathematical Logic*, 2, 1968.
39. L. Zhang. Solving QBF by combining conjunctive and disjunctive normal forms. In *Proc. AAAI'06*.
40. L. Zhang and S. Malik. Towards a symmetric treatment of satisfaction and conflicts in quantified boolean formula evaluation. In *Proc. CP'02*.