JOHANNES KEPLER
UNIVERSITÄT LINZ
Netzwerk für Forschung, Lehre und Praxis

informatik
Schnittstelle Zukunft

An Expansion-based QBF Solver For Negation Normal Form

MAGISTERARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Magisterstudium

INFORMATIK

Eingereicht von:
Florian Lonsing, 0255959

Angefertigt am:
Institut für Formale Modelle und Verifikation

Betreuung:
Univ.-Prof. Dr. Armin Biere

Linz, Dezember 2007

Abstract

The topic of this master thesis 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 costs. The variable with the smallest estimated costs 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 show that Nenofex indeed exceeds a given memory limit less frequently than a resolution-based QBF solver for CNF, but also that there is still room for runtime improvements.

Kurzfassung

In dieser Arbeit wird *Nenofex*, die Implementierung eines Entscheidungsverfahrens für Quantifizierte Boolesche Formeln (QBF) in Negationsnormalform (NNF), beschrieben. In Nenofex werden Variablen sukzessive mittels Expansion aus der Formel eliminiert. Die Elimination von existentiell quantifizierten Variablen durch Resolution ist für Formeln in Konjunktiver Normalform (KNF) im ungünstigsten Fall mit einer quadratischen Zunahme der Formelgröße verbunden. Im Gegensatz dazu führt die Elimination solcher Variablen durch Expansion auf NNF höchstens zu einem linearen Größenwachstum. Diese Eigenschaft dient als Motivation für den Einsatz von Expansion auf NNF. Eine Formel in NNF ist in Nenofex als Baum repräsentiert, dessen Struktur bestimmten Einschränkungen unterliegt. Diese haben den Zweck, die Größe des Baumes sowie die Abstände zwischen den Knoten und der Wurzel gering zu halten. Die Reihenfolge der Expansionen wird nach Schätzung der Expansionskosten festgelegt. Jene Variable mit den geringsten Kosten wird zuerst expandiert. Um Redundanz aus der Formel zu entfernen, wurden vereinfachte Varianten zweier Ansätze aus dem Gebiet der Schaltkreisoptimierung integriert. In Testläufen zeigt sich, dass Nenofex im direkten Vergleich mit einem Verfahren, welches Resolution auf KNF anwendet, zwar weniger häufig ein vorgegebenes Speicherlimit überschreitet, aber hinsichtlich Laufzeit noch Raum für Verbesserungen lässt.

Contents

Chapter 1

Introduction

QBF is the decision problem of quantified boolean formulae. It is a generalization of SAT, the satisfiability problem of propositional logic, where, for a given formula ϕ, it has to be decided if there exists a variable assignment such that ϕ is satisfied. Although quantification is not part of the logic, variables in the context of SAT can be regarded to be existentially quantified. This is reflected in the question whether there *exists* a satisfying truth assignment.

Concerning QBF, variables are either existentially or universally quantified. In contrast to SAT, the question is no longer if a formula *can* be satisfied, but if it is the case that it *is* satisfied with respect to the set of quantified variables.

QBF is the canonical problem in PSPACE, the class of decision problems which are deterministically solvable in polynomial space. Apart from classical results in complexity theory (see [GJ79] for a comprehensive discourse), QBF has found growing attention in the domain of model checking and verification. In the following, some popular approaches are briefly introduced in order to point out the relevance of QBF in this respect.

In explicit model checking [CE81], the implementation of a system which is modelled as a finite state machine is checked for compliance with its specification by explicit state checking. In order to cope with the huge number of possible system states, state spaces can be represented symbolically [BCM+92], which traditionally had been achieved using Binary Decision Diagrams (BDDs) [Bry86]. BDDs in the context of symbolic model checking may suffer considerable size increase. The next step was to apply SAT techniques instead of BDDs: bounded model checking (BMC) [BCCZ99] relies on discovering length-bounded counterexamples for a certain property of the system to be checked. A propositional formula is constructed which is satisfiable if, and only if, such a counterexample exists. This restricted approach of generating counterexamples in combination with SAT makes bounded model checking incomplete: it can not be shown that the systems fulfills the property, but only the converse.

This is the point where QBF comes into consideration. Encoding BMC problems in QBF rather than in SAT yields a more natural representation because symbolic model checking itself is PSPACE complete [JB07][SC85]. Furthermore, BMC becomes complete when using QBF for encoding [BCCZ99].

The quest for efficient decision procedures for QBF has brought up a variety of solvers and benchmark collections [GNT01b]. QBF solvers differ in their strategy how the problem is tackled. Top-down solvers ([CGS98] or [ZM02a] for example) are related to the classical DPLL algorithm [DLL62] for SAT. In fact, many successful techniques from SAT solving such as elimination of units and pure literals [CGS98], non-chronological backtracking [GNT01a] and learning [ZM02a] have been adapted to QBF. Among solvers based on skolemization, we mention sKizzo [Ben05b] and squolem [JBS$^+$07]. Bottom-up solvers eliminate variables successively at the cost of formula size. For example, Quantor [Bie04] expands universal variables and resolves existential ones (concerning classical resolution, we refer to [Rob65] and [DP60], and for resolution on QBF to [BKF95] and [BL94]).

Most QBF solvers work on formulae in conjunctive normal form (CNF) where quantifiers are grouped in an ordered prefix. An advantage of CNF is that it has, together with related data structures and algorithms, already been widely used in SAT checking.

Our approach is different: the topic of this report is *Nenofex*, a bottom-up solver for QBF which uses negation normal form (NNF) as the underlying formula representation. Nenofex has been implemented in C. The solver is closely related to Quantor with respect to the solving strategy. The crucial difference is, that, instead of resolution and expansion as in Quantor, expansion is the only method for eliminating a variable in Nenofex. This restriction stems from the decision of using NNF rather than CNF: expanding an existential variable on NNF causes the formula to increase less than if that variable had been eliminated by resolution on CNF. Similar to Quantor, Nenofex calls a SAT solver if a QBF contains either only existential or only universal variables. For this purpose, a CNF is generated from the NNF in a linear-time transformation. This way, QBF solvers may also profit from improvements made in SAT solvers. In the following sections, we substantiate our argument for using NNF instead of CNF.

1.1 Motivation

Given a quantified boolean formula $F \equiv R \wedge X_0 \wedge X_1$ in CNF. The clause sets $X_0 = \{c_1, c_2, \ldots, c_n\}$ with $|X_0| = n$ and $X_1 = \{c_{n+1}, c_{n+2}, \ldots, c_{n+p}\}$ with $|X_1| = p$ contain all clauses with negative and positive literals of variable x, respectively. Variable x occurs in $n + p$ clauses. R is the set of clauses which do not contain a literal of x (the notation of clause sets has been adopted from [DP60]). We assume that all variables are existentially quantified.

Variable x may be expanded by the following transformation of formula F:

$$F \equiv F[x/0] \vee F[x/1] \tag{1.1}$$

where the expression $F[x/v]$ denotes the formula obtained from F by substituting value v for every occurrence of x. This yields

$$\begin{aligned} F &\equiv (R \wedge X_0 \wedge X_1)[x/0] \vee (R \wedge X_0 \wedge X_1)[x/1] && (1.2) \\ F &\equiv (R \wedge X_1') \vee (R \wedge X_0') && (1.3) \\ F &\equiv R \wedge (X_0' \vee X_1') && (1.4) \end{aligned}$$

In the clause set X_0' (X_1') all negative (positive) occurrences 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 1.4. Note that the resulting formula is not in CNF any more but in NNF.

Returning to the original formula, x is now eliminated by resolution. The set of resolvents X_r is generated as follows

$$X_r = \{c_{i,j} \mid i = 1, \ldots, n, j = n+1, \ldots, n+p,\ c_{i,j} = (c_i \cup c_j) \setminus \{x, \neg x\}\}$$

where X_r contains $n \cdot p$ resolvents (when keeping trivial clauses). After discarding the original clause sets X_0 and X_1 and adding set X_r, formula F now has the form

$$F \equiv R \wedge X_r \tag{1.5}$$

The resulting formula is in CNF.

Observe that formula 1.5 can be obtained from formula 1.4 by applying distributivity of disjunction over conjunction in $(X_0' \vee X_1')$ which yields *exactly* the same set of clauses as X_r. Thus resolution of some variable x has the same effect as expansion if the resulting formula is transformed back to CNF. In both cases, the cost of elimination is the same: $n \cdot p$ clauses will be added to the formula.

If only expansion is carried out, then there is no need to transform formula 1.4 back to CNF, it can be left in NNF. For any arbitrary formula in NNF, expansion of a variable will always yield a formula which is in NNF again, since negation is never applied. It follows from the rule for expansion in equation 1.1 that the size of the formula can at most double, whereas resolution can lead to a quadratic increase in the worst case.

It is exactly this observation which motivates the use of NNF as the underlying 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. In the prior example, if variable x was universally quantified (details of universal expansion will be given in chapter 2), the result of expanding x would be

$$\begin{aligned} F &\equiv (R \wedge X_0 \wedge X_1)[x/0] \wedge (R \wedge X_0 \wedge X_1)[x/1] \quad (1.6)\\ F &\equiv (R \wedge X_1') \wedge (R \wedge X_0') \quad (1.7)\\ F &\equiv R \wedge (X_0' \wedge X_1') \quad (1.8)\end{aligned}$$

where formula 1.8, in contrast to formula 1.4, is in CNF.

In the following section, a concrete example is provided in order to point out the difference in formula sizes between expansion on NNF and resolution on CNF.

1.2 Example: Resolution vs. Expansion

Let F be a formula of the form $R \wedge X_0 \wedge X_1$, where all variables are existentially quantified. The sets R, X_0 and X_1, where $|X_0| = 3$ and $|X_1| = 3$, contain the following clauses:

R :	X_0 :	X_1 :
$(a \vee b)$	$c_1 : (\neg x \vee c \vee \neg d)$	$c_4 : (x \vee f \vee \neg g)$
	$c_2 : (\neg x \vee d \vee \neg e)$	$c_5 : (x \vee g \vee \neg h)$
	$c_3 : (\neg x \vee e \vee \neg c)$	$c_6 : (x \vee h \vee \neg f)$

Resolving variable x generates the set of resolvents X_r, where $|X_r| = 3 \times 3 = 9$. X_r contains the following clauses:

$$\begin{array}{lll} c_{1,4} : (c \vee \neg d \vee f \vee \neg g) & c_{2,4} : (d \vee \neg e \vee f \vee \neg g) & c_{3,4} : (e \vee \neg c \vee f \vee \neg g)\\ c_{1,5} : (c \vee \neg d \vee g \vee \neg h) & c_{2,5} : (d \vee \neg e \vee g \vee \neg h) & c_{3,5} : (e \vee \neg c \vee g \vee \neg h)\\ c_{1,6} : (c \vee \neg d \vee h \vee \neg f) & c_{2,6} : (d \vee \neg e \vee h \vee \neg f) & c_{3,6} : (e \vee \neg c \vee h \vee \neg f)\end{array}$$

Eliminating x by resolution yields

$$\begin{array}{l} (a \vee b) \wedge \\ (c \vee \neg d \vee f \vee \neg g) \wedge (c \vee \neg d \vee g \vee \neg h) \wedge (c \vee \neg d \vee h \vee \neg f) \wedge \\ (d \vee \neg e \vee f \vee \neg g) \wedge (d \vee \neg e \vee g \vee \neg h) \wedge (d \vee \neg e \vee h \vee \neg f) \wedge \\ (e \vee \neg c \vee f \vee \neg g) \wedge (e \vee \neg c \vee g \vee \neg h) \wedge (e \vee \neg c \vee h \vee \neg f)\end{array}$$

Eliminating x by expansion according to formulae 1.1 to 1.4 yields

$$(a \vee b) \wedge (\ \underbrace{((c \vee \neg d) \wedge (d \vee \neg e) \wedge (e \vee \neg c))}_{X_0'} \vee \underbrace{((f \vee \neg g) \wedge (g \vee \neg h) \wedge (h \vee \neg f))}_{X_1'}\ )$$

which is much smaller than the formula obtained by resolution.

1.3 Goals

In order to implement our approach in a new QBF solver, the following problems have to be considered:

- **formula representation:** how can a formula in NNF be compactly represented and what algorithms and data structures are needed for variable expansions and maintenance of the representation?
- **expansion:** how exactly must expansion be performed in order to be able to profit from the property of less size increase which is inherent in NNF?
- **variable scoring:** what are the costs of expanding a variable on NNF and how can they be computed?
- **redundancy removal:** what methods can be applied to remove redundancy produced during expansions?
- **solving strategy:** how can the aforementioned points be integrated in a QBF solver and what heuristics can be applied in order to choose a variable for expansion?

1.4 Outline

The goals listed above could act as milestones in a bottom-up approach for implementing a QBF solver. It has been chosen to take this scheme and use it as a basis for the description of Nenofex. The view on the implementation will be kept abstract in general, but details will be given whenever it is appropriate. In each chapter, only those aspects of the implementation will be considered which are of interest within the respective topic.

This text is organized as follows: in chapter 2, quantified boolean formulae and related concepts are introduced formally. Chapter 3 presents the basic data structures and maintenance algorithms for representing a formula in NNF. A formula is represented as a tree, the structure of which is restricted in order to keep its size small and the distance between nodes and the root short. The focus is put on explaining and justifying design decisions that have been made as they have influence on other functional parts of the solver. A first, abstracted view on the implementation is given.

The central operation of expansion on NNF is the topic of chapter 4. Based on the data structures from the previous chapter, an algorithm is presented in order to identify the parts of the formula which are relevant for expanding a particular variable. Copying unnecessary parts should be avoided.

In chapter 5, a scoring policy is defined in order to rank variables accoring to their predicted expansion costs. Both score computation and a strategy for updating scores is introduced. In Nenofex, generally variables with lowest predicted costs are expanded first in order to keep the formula small.

Chapter 6 deals with the problem of redundancy removal on NNF. Limited versions of two approaches from the domain of circuit optimization and automatic test pattern generation (ATPG) have been integrated in Nenofex. Implementation-related details are mentioned as well.

Finally, in chapter 7 an integrated view of Nenofex is provided. After an explanation of the system and core algorithm, experimental results are considered where Nenofex is compared against Quantor, a CNF-based solver which applies resolution. Chapter 8 summarizes all aspects related to Nenofex and its functional parts.

Chapter 2

Preliminaries

The purpose of this chapter is to formally introduce quantified boolean formulae. Apart from syntax and semantics, related concepts such as subformulae, normal forms or optimizations like unit literals or pure literals are defined.

2.1 Propositional Logic

Definitions have been selected from [HS04], [BL94] and [HR04].

2.1.1 Syntax

Let $Var = \{x_i \mid i \in \mathbf{N}\}$ be a set of propositional variables , $C = \{\mathbf{true}, \mathbf{false}\}$ be the set of truth constants and $O = \{\vee, \wedge, \neg\}$ be the set of boolean operators disjunction ("or"), conjunction ("and") and negation ("not"). Negation is a unary operator, disjunction and conjunction are binary operators. The alphabet A of propositional logic is defined as $A = Var \cup C \cup O \cup \{(,)\}$.

The set of (well-formed) *propositional formulae* is defined inductively as follows:

- the truth constants **true** and **false** are propositional formulae
- every propositional variable is a propositional formula
- if ϕ is a propositional formula then $\neg\phi$ is also a propositional formula
- if ϕ_1 and ϕ_2 are propositional formulae then $(\phi_1 \vee \phi_2)$ and $(\phi_1 \wedge \phi_2)$ are also propositional formulae

Any other string over alphabet A is not a propositional formula.

A *literal* is either a propositional variable x (positive literal) or its negation $\neg x$ (negative literal). An *occurrence* of a variable x is a literal of x. Two literals are complementary if they belong to the same variable and one is positive, the other one negative. Given a propositional formula ϕ, $V(\phi)$ is the set of all variables which have occurrences in ϕ.

2.1.2 Semantics

A variable assignment I of a formula ϕ is a mapping $I : V(\phi) \rightarrow \{\textbf{true}, \textbf{false}\}$ from the set of variables appearing in ϕ to truth values. The value of a formula ϕ under a variable assignment I, written as $\mathit{Val}(\phi, I)$, is defined with respect to the syntactic structure of ϕ as follows:

- $\mathit{Val}(\textbf{true}, I) := \textbf{true}$
- $\mathit{Val}(\textbf{false}, I) := \textbf{false}$
- $\mathit{Val}(x_i, I) := I(x_i)$ where x_i is a variable
- $\mathit{Val}(\neg(\phi), I) :=$ **true** if $\mathit{Val}(\phi, I) =$ **false** and **false** otherwise
- $\mathit{Val}((\phi_1 \vee \phi_2), I) :=$ **true** if $\mathit{Val}(\phi_1, I) =$ **true** or $\mathit{Val}(\phi_2, I) =$ **true** and **false** otherwise
- $\mathit{Val}((\phi_1 \wedge \phi_2), I) :=$ **true** if $\mathit{Val}(\phi_1, I) =$ **true** and $\mathit{Val}(\phi_2, I) =$ **true** and **false** otherwise

A formula ϕ is *satisfiable* if, and only if, there exists a variable assignment I such that $\mathit{Val}(\phi, I) =$ **true**, otherwise ϕ is *unsatisfiable.* Formula ϕ is a *tautology* if, and only if, for all possible variable assignment I, $\mathit{Val}(\phi, I) =$ **true**, or put another way, ϕ is a tautology if, and only if $\neg\phi$ is unsatisfiable. The decision problem, whether a given formula ϕ is satisfiable or not is called SAT [GJ79].

2.1.3 Normal Forms

Normal forms define a set of structurally restricted formulae. In the following, two normal forms are defined, where the second is used as the underlying formula representation in Nenofex.

CNF

A propositional formula ϕ is in *conjunctive normal form (CNF)* if, and only if, it has the form $\phi = C_1 \wedge C_2 \wedge \ldots \wedge C_n$ where each $C_i = (l_{i_1} \vee l_{i_2} \vee \ldots \vee l_{i_m})$. A disjunction C_i is a *clause* and all l_{i_j} are *literals.* Hence a formula in CNF is a conjunction over disjunctions of literals. A clause which consists of just one single literal is a *unit clause.* In order to check whether a formula in CNF is satisfiable, it suffices to check whether every clause is satisfiable.

NNF

A propositional formula ϕ is in *negation normal form (NNF)* if, and only if the negation operator is applied to literals only and if it is not applied more than once at a time (for example, $\neg\neg x$). Every formula which is in CNF is also in NNF (but not vice versa).

An Example

The propositional formula

$$(\neg\neg a \wedge \neg(b \wedge (c \vee d)))$$

is not in NNF because the negation operator is applied to a conjunction, namely $\neg(b \wedge (c \vee d))$. Applying DeMorgan's law twice yields

$$(\neg\neg a \wedge (\neg b \vee (\neg c \wedge \neg d)))$$

which is a formula still not in NNF. Eliminating double negation at literal a finally yields

$$(a \wedge (\neg b \vee (\neg c \wedge \neg d)))$$

which is a formula in NNF but not in CNF. Any arbitrary formula may be transformed into NNF by successive application of DeMorgan's law and elimination of multiple negation. Continuing the example, the formula is transformed into CNF by applying distributivity of disjunction, which finally yields

$$(a \wedge ((\neg b \vee \neg c) \wedge (\neg b \vee \neg d)))$$

We refer to either [HR04] or [BL94] for an algorithmic description of the transformation of arbitrary formulae into NNF and CNF.

A closer look on the result of the last transformation reveals a slight "inaccuracy": the formula is actually not in CNF when strictly interpreting the structural definition of CNF. While a is a clause, expression $((\neg b \vee \neg c) \wedge (\neg b \vee \neg d))$ is not. Enclosing parentheses around this expression had to be eliminated in order to fulfill the properties of CNF.

2.1.4 Associativity and Arity of Operators

The syntactic definition of propositional formulae requires that the boolean operators disjunction and conjunction are applied to exactly two operands. Hence it is syntactically incorrect to write $(a \vee b \vee c)$ instead of $(a \vee (b \vee c))$ or $((a \vee b) \vee c)$.

However, the last three formulae are equivalent by associativity of disjunction and conjunction. Formulae like

$$(a \wedge ((\neg b \vee \neg c) \wedge (\neg b \vee \neg d)))$$

from the last example may be "flattened" to

$$(a \wedge (\neg b \vee \neg c) \wedge (\neg b \vee \neg d))$$

which finally is in CNF, and thus clarifying the "inaccurracy" mentioned above. Applying associativity corresponds to extending the arity of the binary operators to an arbitrary number of operands which yields *n-ary operators*.

2.1.5 Subformulae

For a given formula ϕ, the set of subformulae of ϕ, written as $\mathit{subf}(\phi)$, is defined recursively as follows:

- if ϕ is of the form **true** then $\mathit{subf}(\phi) := \{\mathbf{true}\}$
- if ϕ is of the form **false** then $\mathit{subf}(\phi) := \{\mathbf{false}\}$
- if ϕ is of the form x_i then $\mathit{subf}(\phi) := \{x_i\}$
- if ϕ is of the form $\neg\phi_1$ then $\mathit{subf}(\phi) := \{\phi\} \cup \mathit{subf}(\phi_1)$
- if ϕ is of the form $(\phi_1 \otimes \ldots \otimes \phi_n)$ where $n \geq 2$ and $\otimes \in \{\vee, \wedge\}$, then $\mathit{subf}(\phi) := \{\phi\} \cup \bigcup_{i=1}^{n} \mathit{subf}(\phi_i) \cup \bigcup \{(\phi_{i_1} \otimes \ldots \otimes \phi_{i_n}) | \text{where } 1 \leq i_1 < \ldots < i_n \leq n\}$

Note that $\mathit{subf}(\phi)$ contains at least one element. Formula ϕ is broken up into its constituent parts, all of which are well-formed formulae. For n-ary conjunctions (disjunctions) over formulae ϕ_i, all possible conjunctions (disjunctions) which can be formed from formulae ϕ_i are subformulae.

For example, for formula ϕ

$$\phi \equiv (a \vee \neg(b \wedge c \wedge d))$$

the set $\mathit{subf}(\phi) = \{(a \vee \neg(b \wedge c \wedge d)), a, \neg(b \wedge c \wedge d), (b \wedge c \wedge d), b, c, d, (b \wedge c),$ $(b \wedge d), (c \wedge d\}$.

2.2 Quantified Boolean Formulae

QBF is a generalization of SAT. This section introduces syntax, semantics and related notions on top of the definitions from the previous section. Our definitions are closely related to [BL94].

2.2.1 Syntax

Alphabet A of propositional logic (section 2.1.1) is extended by adding two new symbols $\exists$ and $\forall$, where the former denotes existential, the latter universal quantification. Hence $A := A \cup Q$ where $Q := \{\exists, \forall\}$ is the set of quantifier symbols (or quantifiers). Based on the given syntactic definition of propositional formulae, the set of quantified boolean formulae (QBF) is defined as follows:

- every propositional formula is a QBF
- for a QBF ϕ and propositional variables x and y, both $\exists x\phi$ and $\forall y\phi$ are QBF
- no other formula is a QBF

Adding quantified variables like $\exists x$ to a QBF ϕ makes sense only if variable x appears in ϕ. The building rules allow QBFs of the form $S_1S_2 \ldots S_n\phi$ only where each S_i stands for either $\exists x_i$ or $\forall x_i$ and where ϕ is free of quantifiers. A formula of this particular syntactic structure is in *prenex normal form*. The sequence $S_1S_2 \ldots S_n$ is the *quantifier prefix* which is put into concrete terms in the following. The definitions have been taken from [Bie04].

Let $S_1S_2 \ldots S_n\phi$ be a QBF. The set of variables in ϕ is partitioned into n sets S_i which are called *scopes*: $V(\phi) = S_1 \cup S_2 \cup \ldots \cup S_n$ and $S_i \cap S_j = \emptyset$ for $i \neq j$ and $i, j \leq n$. A scope S_i is *existential* (*universal*) if it is associated with an existential (universal) quantifier, written as $\mathit{type}(S_i) = \exists$ ($\mathit{type}(S_i) = \forall$). By convention, two adjacent scopes S_i and S_{i+1}, where $i < n$, must not be both existential or both universal. From this follows that scopes may contain more than one variable. A variable $x \in V(\phi)$ is quantified existentially (universally) if it belongs to an existential (universal) scope. Depending on the type of the scope, a variable is either classified as existential or universal. The scope of x is denoted by $\mathit{scope}(x)$. Variables must not appear in the formula without being quantified (so-called free variables) nor be quantified more than once. A sequence of scopes is a linearly ordered set $S_1 < S_2 < \ldots < S_n$ which follows from the order of appearance of the scopes in the quantifier prefix. Scope S_1 is the outermost, scope S_n the innermost scope. Variables x and y are ordered with respect to the scope order. In case that $\mathit{scope}(x) = \mathit{scope}(y)$, an arbitary order between x and y may be chosen.

The QDIMACS format [QDI05] has become the standard format for QBF instances given in CNF. It extends the DIMACS format for SAT [DIM93] by a section that lists the quantifier prefix. Therefore, all QBF solvers which read problem files in QDIMACS format also accept SAT problems in DIMACS format.

2.2.2 Semantics

Let *Val* be a valuation function which maps a given QBF ϕ to either **true** or **false**, where *Val* is defined according to the syntactic structure of ϕ as follows [BL94]:

- $\phi =$ **true** : $\mathit{Val}(\phi) =$ **true**
- $\phi =$ **false** : $\mathit{Val}(\phi) =$ **false**
- $\phi = \neg\psi$: $\mathit{Val}(\phi) =$ **true** if $\mathit{Val}(\psi) =$ **false** and **false** otherwise
- $\phi = \phi_1 \vee \phi_2$: $\mathit{Val}(\phi) =$ **true** if $\mathit{Val}(\phi_1) =$ **true** or $\mathit{Val}(\phi_2) =$ **true** and **false** otherwise
- $\phi = \phi_1 \wedge \phi_2$: $\mathit{Val}(\phi) =$ **true** if $\mathit{Val}(\phi_1) =$ **true** and $\mathit{Val}(\phi_2) =$ **true** and **false** otherwise
- $\phi = \exists x\ \psi$: $\mathit{Val}(\phi) =$ **true** if $\mathit{Val}(\psi[x/0]) =$ **true** or $\mathit{Val}(\psi[x/1]) =$ **true** and **false** otherwise
- $\phi = \forall x\ \psi$: $\mathit{Val}(\phi) =$ **true** if $\mathit{Val}(\psi[x/0]) =$ **true** and $\mathit{Val}(\psi[x/1]) =$ **true** and **false** otherwise

The expression $\phi[x/v]$ represents the formula obtained from ϕ by substituting value v for every occurrence of variable x in ϕ. For example, formula

$$\forall x \exists y\ ((x \vee \neg y) \wedge (\neg x \vee y))$$

is true, whereas

$$\exists y \forall x\ ((x \vee \neg y) \wedge (\neg x \vee y))$$

is false (the scope order has been reversed).

2.2.3 Optimizations

Elimination of unit literals or pure literals plays an important in role in efficient SAT solvers. These techniques have been adapted to QBF [CGS98] as explained in the following.

Unit Elimination

A clause which contains only one single literal is called *unit clause* and the literal a *unit literal*. In order to satisfy a unit clause, the literal must be set to true by assigning the respective variable accordingly. For universal unit literals, unsatisfiability can be concluded immediately since the variable had to be assigned both truth values.

Pure Literals

If a variable either has only positive or only negative literals then these literals are called *pure literals*, *unates* or *monotone literals* [CGS98]. Existential (universal) pure literals can be set to true (false) by assigning the respective variable accordingly.

Forall-reduction

Forall-reduction [BKF95] [Bie04] is an operation which removes universal literals from certain clauses. From a given clause, a universal literal can be removed 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.

This operation is defined to work on CNF and, to our knowledge, it is not clear whether and how it can be applied to formulae in NNF.

2.2.4 Expansion

Given a QBF $S_1 S_2 \dots S_{n-1} S_n\ \phi$ in NNF, expansion is defined for existential variables from scope S_n if $\mathit{type}(S_n) = \exists$, and for universal variables either (1) from scope S_n if $\mathit{type}(S_n) = \forall$ and $\mathit{type}(S_{n-1}) = \exists$, or (2) from S_{n-1} if $\mathit{type}(S_{n-1}) = \forall$ and $\mathit{type}(S_n) = \exists$.

Note that case (1) need not be applied on formulae in CNF since forall-reduction could remove all occurrences of universal variables in S_n instead. Furthermore, applying expansion as defined in a general way below will almost double the size of the formula, which is not desireable. In chapter 4 this problem is investigated in the context of NNF. For expansion on CNF, we refer to [Bie04] or [BB07]. An approach for expanding universal variables from arbitrary scopes is presented in the latter. Our definitions are related to the ones given in [Bie04] and [BB07].

Existential Expansion

Given a QBF

$$S_1 S_2 \dots S_{n-1} S_n\ \phi$$

where $\mathit{type}(S_n) = \exists$, the result of expanding variable x from scope S_n is

$$S_1 S_2 \dots S_{n-1} (S_n \setminus \{x\})\ (\phi[x/0] \vee \phi[x/1])$$

Universal Expansion

The two cases described above are defined as follows:

1. Given a QBF
$$S_1 S_2 \dots S_{n-1} S_n \ \phi$$
where $\mathit{type}(S_n) = \forall$, the result of expanding variable x from scope S_n is
$$S_1 S_2 \dots S_{n-1} (S_n \setminus \{x\}) \ (\phi[x/0] \wedge \phi[x/1])$$

2. Given a QBF
$$S_1 S_2 \dots S_{n-1} S_n \ \phi$$
where $\mathit{type}(S_{n-1}) = \forall$ and $\mathit{type}(S_n) = \exists$, the result of expanding variable x from scope S_{n-1} is
$$S_1 S_2 \dots (S_{n-1} \setminus \{x\})(S_n \cup S'_n) \ (\phi[x/0] \wedge \phi'[x/1])$$
where set S'_n contains duplicated variables v' for each variable $v \in S_n$ and ϕ' is obtained from ϕ by substituting v' for all occurrences of $v \in S_n$.

Duplicating existential variables as defined in case (2) is conservative but pessimistic. In chapter 4 an algorithm is presented for restricting the set of duplicated variables with respect to expansion on NNF. Concerning formulae in CNF, such methods have been proposed in [Bie04], [BB07] or [Ben05b], for example.

Chapter 3

Formula Representation

This chapter introduces *NNF-trees*, a data structure for representing formulae in NNF in Nenofex. An NNF-tree consists of operator- and literal nodes and fulfills certain structural restrictions in order to keep its size small and the distance between nodes and the root short. These two properties, a small tree and and short distances, are particularly important with respect to (still to be introduced) algorithms which operate on the NNF-tree. Furthermore, it is guaranteed that a formula in CNF, which is the standard format for QBF instances [QDI05], has a flat representation as an NNF-tree.

The first section introduces fundamental definitions of graph properties and trees which have been taken from [OW02]. In section 3.2, the structure of NNF-trees is defined which is based on ordinary trees. The basic structure is sufficient for representing formulae in NNF, but neither guarantees small tree sizes nor short distances. For this purpose, three kinds of structural restrictions are introduced in section 3.3, which an NNF-tree must fulfill. Finally, the implementation of NNF-trees and related maintenance algorithms is described in section 3.4.

3.1 Graph Terminology

A directed graph $G = (N, E)$ is defined by a set of nodes N and a set of directed edges $E \subseteq N \times N$. There is an edge from a node n_1 to another node n_2 if $(n_1, n_2) \in E$. If there is an edge from n_1 to n_2 then the two nodes are *adjacent*. A node n_2 is a *successor* of another node n_1 if there exists a sequence of nodes $p_1, p_2, \ldots, p_n$ such that $n_1 = p_1$ and $n_2 = p_n$ and $(p_i, p_{i+1}) \in E$ for all i where $1 \leq i < n$. In this case n_2 is *reachable* from n_1. The sequence of nodes $p_1, p_2, \ldots, p_n$ is a *path* from n_1 to n_2 with length $n-1$. In case that $n = 1$, the path is trivial and has length zero. If node n_2 is a successor of node n_1 then node n_1 is a *predecessor* of node n_2. Concerning trivial paths, every node n is a successor and predecessor of itself (a trivial successor or predecessor). Node n_1 is an *immediate predecessor* of node n_2 if the path from n_1 to n_2 has length one. A *subgraph* S_G of graph G is defined by a set of nodes $N' \subseteq N$ and a set of edges $E' = E \cap (N' \times N')$. A *directed acyclic graph (DAG)* is a directed graph which does not contain cycles. A *cycle* is a path $p_1, p_2, \ldots, p_n$ such that $p_1 = p_n$.

A (directed) *tree* is a DAG where each node has exactly one immediate predecessor, except one unique node called *root* which has no predecessor. If there is an edge from node n_1 to node n_2 then n_1 is the *parent* of n_2, written as $\mathit{parent}(n_2) = n_1$, and n_2 is a *child* of n_1. All nodes in the tree are reachable from the root via a unique path and have exactly one parent, except the root which has no parent. From this property it follows that for each node there exists a unique path to the root if edges are inverted. Each node n has (or is at) a *level*, written as $\mathit{level}(n)$, which is the length of the path from the root to n. The root of a tree has level 0 and an arbitrary node n has level $\mathit{level}(\mathit{parent}(n)) + 1$. Nodes which have no children are external nodes or *leaf nodes*. All other nodes are *internal nodes*. Two nodes are *siblings* if they have the same parent.

A *common ancestor* of a pair of nodes (n_1, n_2) in a tree is a node which is a predecessor of both n_1 and n_2. The *least common ancestor (LCA)* of a pair of nodes, written as $\mathit{lca}(n_1, n_2)$ where $\mathit{lca} : N \times N \rightarrow N$ is a mapping, is their common ancestor with maximum level, that is, which is farthest away from the root of the tree. Hence there is no child of $\mathit{lca}(n_1, n_2)$ which is a common ancestor of (n_1, n_2). When regarding lca as an operator, it can be observed that it has the following properties:

- commutativity: $\mathit{lca}(n_1, n_2) = \mathit{lca}(n_2, n_1)$
- associativity: $\mathit{lca}(n_1, \mathit{lca}(n_2, n_3)) = \mathit{lca}(\mathit{lca}(n_1, n_2), n_3)$

Applying these properties, the definition of lca is extended from pairs to sets of nodes in the following way:

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

For the tree example in figure 3.1:

- the set of internal nodes is $\{1, 2, 3, 4, 7\}$
- the set of leaf nodes is $\{5, 6, 8, 9, 10, 11, 12, 13\}$
- node 6 is child of node 3
- node 3 is parent of node 6, 7, 8 and 9

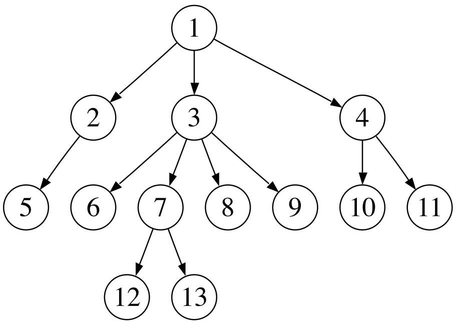


Figure 3.1: A basic tree example

- the set of successors of node 3 is $\{3, 6, 7, 8, 9, 12, 13\}$
- the set of predecessors of node 12 is {12, 7, 3, 1}
- the sequence $(1, 4, 10)$ forms a path of length 2 from node 1 to node 10
- node 1 is the root of the tree and the only node which has level 0
- nodes 5, 6, 7, 8, 9, 10 and 11 all have level 2
- from node 2, nodes 2 and 5 are reachable
- nodes 4 and 1 are all common ancestors of nodes 10 and 11
- $lca(12, 13)$ is 7, $lca(5, 6)$ is 1, $lca(1, 12)$ is 1 and $lca(6, 8, 13)$ is 3
- nodes $N' = \{4, 10, 11\}$ and edges $E' = \{(4, 10), (4, 11)\}$ form a subtree

3.2 Representing a Formula in NNF

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 nodes* N_L, that is $N = N_O \cup N_L$ and $N_O \cap N_L = \emptyset$. A node $n \in N$ belongs to exactly one of the sets N_O and N_L. The set N_O (N_L) comprises exactly the set of internal nodes (leaf nodes) of the tree.

The set N_O is further 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. Hence an operator node denotes an n-ary boolean function where n is the number of its children. A node $n \in N_O$ belongs to exactly one of the sets $N_\vee$ and $N_\wedge$.

A node $n \in N_L$ denotes one single occurrence of some variable $x \in V$. It either represents a positive or a negative literal of x. Conversely, an occurrence of some variable x is represented by exactly one node $n \in N_L$. Analogously to the notion of (positive or negative) literals of a variable, there is the notion of (positive or negative) literal nodes of a variable. The least common ancestor (LCA) of a variable $x \in V$, written as $lca(x)$, is the LCA over all of its occurrences. The membership of a node n in either N_L or $N_\vee$ or $N_\wedge$ is regarded as the *type* of n, written as $\mathit{type}(n) = t$ where $t \in \{$*literal node, OR-node, AND-node*$\}$.

Each node $n \in N$ denotes a subformula (see section 2.1.5 on page 10) of ϕ. The subformula of a literal node n_l is the single literal represented by n_l. The subformula of an OR-node (AND-node) is the logical disjunction (conjunction) over the subformulae of all of its children. The subformula of node n corresponds to the subgraph (subtree) with root n. Every subtree denotes a subformula of ϕ and vice versa. Note that, while every node corresponds to a subformula, the converse is not true (for example, for a subformula $(\phi_{i_1} \vee \ldots \vee \phi_{i_n})$ which is formed from a subset of ϕ_i in an n-ary disjunction $(\phi_i \vee \ldots \vee \phi_n)$, where $1 \leq i_1 < \ldots < i_n < n$).

Truth constants do not have an explicit representation as nodes because they can always be eliminated from a given formula by applying one of the following equivalences as appropriate:

$$\begin{aligned}(a \wedge \mathbf{true}) &= a\\(a \vee \mathbf{true}) &= \mathbf{true}\\(a \wedge \mathbf{false}) &= \mathbf{false}\\(a \vee \mathbf{false}) &= a\end{aligned}$$

Hence the presence of truth constants in a formula indicates redundancies which are not represented in the graph of the formula.

The tree in figure 3.2 shows a representation for the formula

$$a \wedge (\neg b \vee (c \wedge d)) \wedge (\neg a \vee (b \wedge d \wedge \neg e) \vee (a \wedge \neg d))$$

and at the same time introduces the basic graphical notation that is used throughout illustrations of formula trees: an AND-node (OR-node) is represented as a triangle △ (inverted triangle ▽) resembling the symbol for logical conjunction $\wedge$ (disjunction $\vee$), and a literal node as a box □. Diamond-shape nodes in two styles ◇ and ⟡ as in figure 3.9 (page 32) both may either stand for an AND-node or an OR-node, yet two adjacent diamond-shape nodes have different types and styles. A circle ∘ at the end of an edge to a literal node denotes the negation operator. The leftmost (rightmost) child in the set of children of an operator node is considered as the first child (last child) of that node. Nodes in the graph may

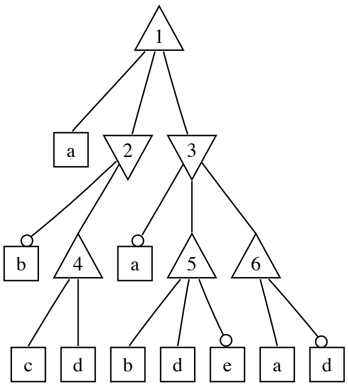


Figure 3.2: Graph of formula $a \wedge (\neg b \vee (c \wedge d)) \wedge (\neg a \vee (b \wedge d \wedge \neg e) \vee (a \wedge \neg d))$

be labelled arbitrarily except literal nodes which always carry the label of their variable $(a, b, c, \ldots)$. A bigger triangle with double peripheral lines like in figure 3.5 (page 24) denotes an arbitrary subgraph.

For the tree in figure 3.2

- the sets of operator nodes are $N_O = \{1, 2, 3, 4, 5, 6\}$, $N_\vee = \{2, 3\}$ and $N_\wedge = \{1, 4, 5, 6\}$ and N_O is at the same time the set of internal nodes
- the types of nodes 1 and 2 are *AND-node* and *OR-node*, respectively
- variable b has a negative occurrence at node 2 and a positive one node 5
- the subformula of the single literal at node 3 is $\neg a$
- the subformula of node 5 is $(b \wedge d \wedge \neg e)$, the corresponding subgraph (N', E') is defined by $N' = \{5, b, d, \neg e\}$ and $E' = \{(5, b), (5, d), (5, \neg e)\}$
- the subformula of node 6 is $(a \wedge \neg d)$
- the subformula of node 3 is $(\neg a \vee (b \wedge d \wedge \neg e) \vee (a \wedge \neg d))$ which is a disjunction over the subformulae of its children $\neg a$, 5 and 6
- the arity of the boolean function denoted by node 3 is 3, which corresponds to the number of its children
- the LCA of variable d is node 1

3.2.1 Tree vs. DAG

DAGs would have been an alternative approach for representing a formula in NNF. In a DAG, nodes might have more than one parent. Such nodes are considered to be *structurally shared* among their parents. A well-known, DAG-related formula representation are And-Inverter Graphs (AIGs) [KPKG02] where the set of boolean operators used in the representation is restricted to binary conjunction and negation. Methods for identifying possible structural sharing in AIGs have been studied in [BB06], [BB04].

To our knowledge, structural sharing in combination with n-ary operators has not been studied at a large extent, but obviously there is much more complexity involved with this respect.

Furthermore, trees as proposed in the previous section allow a CNF to be represented in a natural way: each literal node in the tree corresponds to exactly one literal in the CNF, each OR-node to exactly one clause and the single AND-node at the root to the conjunction over the clauses.

If DAGs had been used, then the set of parents of each node needs to be stored and maintained under a sequence of graph modifications. This is another source of complexity. Furthermore, in a DAG a node may be child of more than one parent, which complicates the implementation of child sets.

Finally, at a later stage of development it turned out that the structurally inherent tree property of nodes to have exactly one parent has a positive influence on the complexity of algorithms related to redundancy removal (see chapter 5).

3.3 Structural Restrictions

In addition to the aforementioned basic properties of a tree which represents a formula in NNF, restrictions are imposed on the structure of the graph of the following kind:

- *arity of operators:* the number of children of an operator node determines the arity of the boolean function denoted by that operator node

- *alternating types over levels:* the type of the children of a node is restricted with respect to the parent's type

- *one-level-simplification:* the set of literal nodes which may occur in the set of children of a node is restricted

The tree representation that has been defined above in section 3.2 is sufficient in order to represent an arbitrary formula. However, one source of motivation for restricting the structure of the tree stems from expected positive effects on the

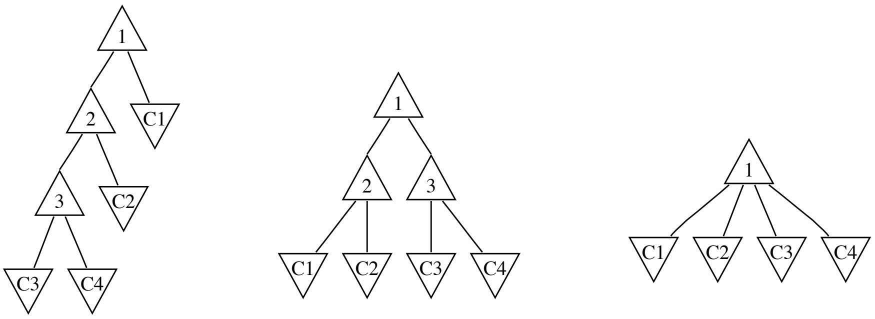


Figure 3.3: Three representations of one and the same formula in CNF

runtime of particular maintenance algorithms: whenever the tree is modified, for example by inserting or deleting subgraphs, then certain properties have to be kept up to date for each node. The drawback comes with an increased implementation complexity because every transformation of the graph has to be carried out in accordance with the restrictions. This requirement affects all functional parts of the solver. Another positive effect is that the restricted structure guarantees a flat representation for CNFs.

3.3.1 Arity of Operators

An operator node may have an arbitrary number of children, yet must have at least two. A literal node has no children. The number of children of an operator node $n_o \in N_O$ corresponds to the arity of the logical function denoted by n_o.

The effect of this restriction on the structure of the tree is twofold. First, the number of required operator nodes is reduced and the distance between literal nodes and the root, the level, is kept small. Second, operator nodes may have large sets of children. While the first effect can be regarded as entirely positive, because the tree becomes compact, the second is problematic in an implementation-related sense. If only a small subset of children needs to be accessed, then fully inspecting a large set of children is inefficient. A solution to this problem is presented in sections 3.4.4 (page 30) and 3.4.6 (page 33).

Figure 3.3 shows three different trees of one and the same (arbitrary) formula in CNF with four clauses as they could look like with different arities of operators. OR-nodes are labelled $C_1, C_2, \ldots$ in order to indicate that they represent whole clauses (for reasons of brevity, literal nodes have been omitted). The figures show the formula in its original form, that is before any modifications have been carried out.

In the leftmost tree, operator nodes are binary. All literal nodes in clauses C_3 and

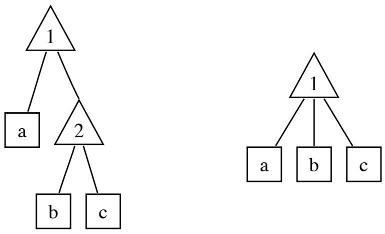


Figure 3.4: Enforcing alternating node types

C_4 are at level four. The maximum node level in a tree of almost linear shape like this will be the higher the more clauses are present.

In the second tree, operators are still binary but the tree has been balanced by interchanging nodes C_1 and 3 to keep levels small. Thus the maximum level has been decreased to three.

Finally, in the third tree, the arity of operators is arbitrary which yields a tree where, for a formula in CNF, *all* literals have a level of two, except unit literals which have a level of one. This situation can never be achieved when the arity is fixed, not even with balancing (provided a sufficiently large number of clauses compared to the arity). Note that, like in the third picture, the tree for a given CNF with n clauses will have exactly one AND-node (root) and exactly n OR-nodes (clauses).

Hence when allowing operators to have arbitrary arity, the effect of reduced node levels comes from the structural property itself and does not require explicit, and possibly complicated, balancing.

Regarding the requirement of a minimum arity of two, the only time when the tree sructure has to be repaired is when an operator node has only one child left. This problem is called *parent merging* and is the topic of section 3.4.5 (page 32). The following restriction additionally affects node levels and the number of required operator nodes positively.

3.3.2 Alternating Types Over Levels

An operator node $n_o \in N_\vee$ ($n_a \in N_\wedge$) may only have literal nodes or AND-nodes (OR-nodes) as children. Hence for the path $p_1, p_2, \ldots, p_n$ from $root = p_1$ to a literal node $n_2 = p_n$ the following property holds: $type(p_i) \neq type(p_{i+1})$ for $1 \leq i < n$. In other words, any operator node in the tree may only have children of a different type. This rule can be regarded as applying associativity whenever possible. For example, in the representation of formula $(a \wedge (b \wedge c)) = (a \wedge b \wedge c)$ in figure 3.4 in the left tree, the violation of this restriction can be rectified by applying associativity of conjunction, yielding the situation as shown on the right.

Note that n-ary operators are a requirement for this restriction in order to be carried out but not vice versa. In other words, if the arity is fixed then situations in which two operator nodes of same type are in a parent-child relationship can in general not be avoided (see the second tree in figure 3.3, for example). On the other hand, if operator nodes had arbitrary arity but alternating types were not enforced, then there would be no guarantee that a CNF representation is flat like the third tree in figure 3.3. In a straightforward way this flat tree can be obtained by adding one clause after the other to the root. In contrast, a tree for $((C_1 \wedge C_2 \wedge \ldots \wedge C_k) \wedge (C_{k+1} \wedge C_{k+2} \wedge \ldots C_n))$ where C_i are clauses is not guaranteed to be flat when not requiring alternating types over levels even if operators have arbitrary arity.

The two restrictions described so far ensure that a given QBF in CNF *has*, not just *can* have, a compact representation in our NNF-oriented tree structure and that operator nodes are saved whenever associativity can be applied. Furthermore, node levels are kept low. The next structural restriction again aims at saving nodes but this time by identifying logical redundancies within the set of children of an operator node.

3.3.3 One-level Simplification

An operator node may have as a child at most one literal node which denotes an occurrence of one and the same variable $x \in V$. Hence if an operator node has as a child a literal node of some variable x then it must neither have another positive literal node of x nor a negative literal node of x as a child. Multiple occurrences of one and the same variable as children of the same operator node *always* indicate the presence of redundant nodes in the graph. The set of possible redundancies can be be expressed by the following equivalences:

$$\begin{aligned}(a \vee a) &= a \\ (a \vee \neg a) &= \textbf{true} \\ (a \wedge a) &= a \\ (a \wedge \neg a) &= \textbf{false}\end{aligned}$$

Identical literals can be removed from the set of children. In case of complementary literals, the boolean function denoted by their parent collapses to a constant value. Constant parent nodes are deleted immediately after detection.

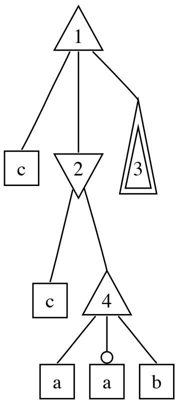


Figure 3.5: One-level redundancy

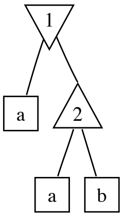


Figure 3.6: Two-level redundancy

For example, figure 3.5 shows a situation where complementary literals of variable a at node 4 induce redundancy (node 4 is false). If node 4 is deleted then its parent (node 2) will only have one child left. This situation requires parent merging, which is described in section 3.4.5 in detail.

The method for removing simple redundancies is called one-level-simplification because it can be enforced by checking the children of an operator node only which, by definition, all have the same level. It is not necessary to consider grandchildren, however further redundancies may be detected when checking successor nodes over more levels. Figure 3.6 shows a simple two-level redundancy which can not be detected by inspecting all children of a single operator node. In this example, the given formula $(a \vee (a \wedge b))$ is equivalent to a. Redundancy removal on NNF is the topic of chapter 6.

All structural restrictions defined in this section can be maintained locally at the regions of the NNF-tree where modifications have been performed. It is not necessary to traverse the tree.

3.4 Implementation

This section introduces fundamental data structures for representing QBFs in NNF. The low-level design and implementation follows the formal graph definitions which were given in the previous section very closely. Apart from data structures (sections 3.4.1 and 3.4.2), basic algorithms for maintaining a formula representation under a set of basic operations such as deleting and inserting nodes or assigning variables are presented (section 3.4.4).

According to the definitions of propositional formulae, QBF and trees, a representation of a QBF in NNF needs to store the prefix in terms of scopes and their variables on the one hand, and the formula in terms of a tree consisting of operator and literal nodes on the other hand.

3.4.1 Prefix

This section describes the data structures for representing the prefix of a QBF.

Scopes

The prefix is represented as an ordered list of n scope objects where n is the total number of scopes in the prefix. Each scope object has a type (either existential or universal) and a positive integer *nesting* which corresponds to the position of the scope in the ordered scope list. Hence the outermost scope has nesting 1, the innermost scope nesting n. Variables in a scope are represented as variable objects which are stored in a *variable list*. The scope object has a pointer to this list.

Variables and Literals

Each variable object has a unique positive integer *id*, a positive and negative literal object and a pointer back to the scope object it belongs to. A literal object is embedded in a variable object and has a constant-value tag indicating whether it is negated or not and a pointer back to the variable it belongs to. The purpose of this pointer is to access the variable from within one of its two literal objects. All positive (negative) literals of a particular variable are stored in an occurrence list which is accessible from the respective positive (negative) literal object. An occurrence list is implemented as a doubly linked list, which allows insertion and deletion of entries in constant time. Links to the previous and next occurrence are embedded in a literal node object (see next section). Hence a literal object additionally has pointers to the first and last entries in the occurrence list and a counter which corresponds to the current number of list entries. Whenever an entry is added or removed from the list, the counter is updated accordingly.

Figure 3.7 illustrates the relations between scopes, variables and their literal objects. The bar on top represents the list of scopes. In this example, the outermost scope is existential, the values for type and nesting are set accordingly, and the second scope is universal. A scope has n variables in a list (bar in the middle) which is pointed to be the *vars*-pointer. The fields of a variable object are shown in the bar at the bottom. The two columns labelled 'pos_lit' and 'neg_lit' represent the two embedded literal objects of a variable. The tags which indicate

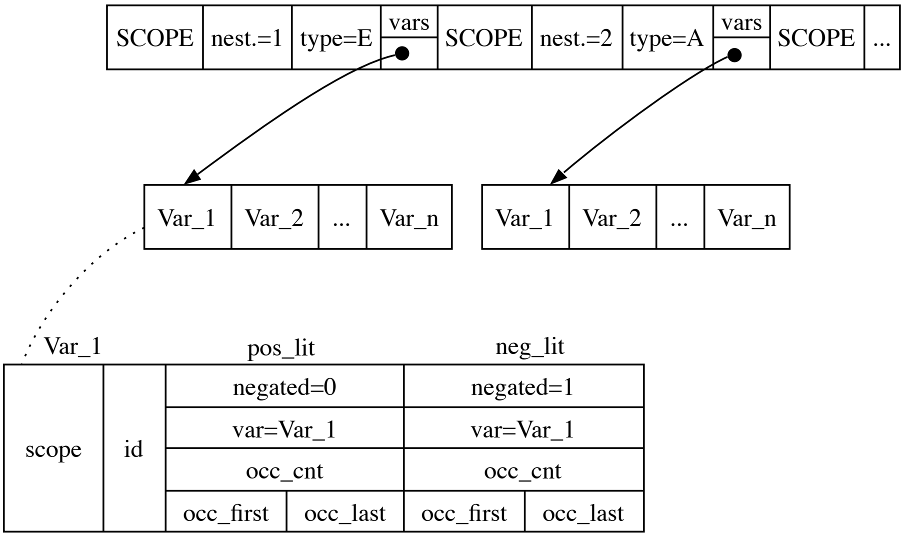


Figure 3.7: Prefix representation

negation are set to constant values. The pointer from each literal object back to the variable is not shown, but indicated by the entry $var = Var\_1$.

3.4.2 Formula

A formula is represented as a tree of node objects which is accessible via a pointer to the root of the tree. Depending on its type, a node object stores different pieces of information. First, a node object has, regardless of the node type, a unique positive integer *id*, a non-negative integer *level* denoting the distance of the node to the root, a type tag (*literal node, OR-node, AND-node*), a pointer to its parent (which is set to null in the tree root), a non-negative integer *subformula-size* which corresponds to the number of successors of that node (always one for literal nodes) and pointers to the previous and next sibling which are used to link all children of an operator node. The subformula size of a node corresponds to the number of nodes in the subgraph rooted at that node. It is computed by summing up the subformula sizes of the children and adding one for the node itself.

The children are stored in child lists which are, similar to occurrence lists, implemented as doubly linked lists with a counter for storing the current length of the list. Therefore an operator node object additionally has pointers to its first and last child. Together with the embedded sibling-pointers (also called “level links”)

Field	*Relevance*
id	O,L
level	O,L
type	O,L
parent	O,L
level links	O,L
child list	O
occ.links	L
lit	L
child count	O
subformula size	O,L

Table 3.1: Relevant node fields with respect to node type

in an operator node's children, these pointers form the child list of an operator node. Note that this design of child lists is not possible if DAGs are used, since *embedded* sibling pointers require a node to occur in exactly one child list. In DAGs, this is generally not the case.

Finally, a literal node object has additional pointers to the previous and next occurrence in the occurrence list (also called "occurrence links") and a pointer to the corresponding (positive or negative) literal object of the variable they belong to, which is called *literal pointer*. A literal node object denotes a negative (positive) occurrence of a variable if the literal pointer points to the negative (positive) literal object of the variable.

Table 3.1 provides a summary of the pieces of information that are stored in nodes. In the second column, "O" ("L") means that the respective field is relevant for operator nodes (literals nodes). In our implementation in C, there is only one single coarse-grain structure representing a node object. Operator and literal nodes are distinguished solely by dereferencing their type tag. Unused fields like "lit" for an operator node are set to null. Optimizing this design for memory has not been considered so far.

Node Marks and Node IDs

In the given implementation-related description of a prefix and a formula, the possibility of marking objects has been neglected. In fact, it is often necessary to mark nodes or variables within (still to be described) maintenance algorithms after the graph of the formula has been modified. In order to keep the view on the implementation abstract, the presence of certain *object marks* is taken for granted whenever marking is needed. Marks, their names and meanings are introduced on demand. In a description of an algorithm it is assumed that, unless otherwise stated, all marks of whatever name are cleared.

The ID of a node or a variable plays a minor role in Nenofex. IDs could always be assigned arbitrarily on demand, provided that each variable and operator node gets a unique ID and each positive (negative) literal node the same ID (negated ID) as its variable. Hence a negative literal node of variable x would get $-id(x)$. The relation between a variable, its literals and the respective occurrences is entirely expressed with pointers as described and does not rely on IDs.

3.4.3 Low-level Pointers: A Comprehensive Example

This section considers a comprehensive example illustrating the low-level pointer organization. Figure 3.8 shows pointers between the nodes of the tree representation for formula $(a \wedge \neg b \wedge (a \vee c))$. Different styles (solid or dashed) or labels indicate different purposes of pointers. Pointers of a particular kind are labelled once only in order to keep the picture clear. Furthermore, null pointers are not indicated in the picture.

A node in the picture has five fields: type, id, level, number of children, and subformula size. Node IDs are assigned arbitrarily except for literal nodes which have as an ID the name of the variable they belong to. The number of children in literal nodes (nodes with type "LIT") is always zero, the subformula size always one because of the trivial subformula which is the literal denoted by the literal node itself. Node 1 is the root and has a subformula size of 6. All children of an operator node (nodes 1 and 2) have the same level.

Dashed pointers are parent pointers (label "par"). Node 1 is the root and is the only node in the tree which has no parent pointer.

Solid pointers which start in an operator node and which point to the first and last child of that operator node are one part of the child list implementation (labels "cfirst" and "clast"). Its second part is formed by horizontal solid pointers which doubly link the children of an operator node (labels "cprev" and "cnext").

The variable table in the picture is abstract and compactly summarizes the illustrations from figure 3.7, so details with this respect are ignored in this example. The table contains three variables, each of which has its positive (negative) literal object on top (bottom) of the respective column.

Dotted pointers represent the implementation of occurrence lists. A literal object in the abstract variable table has two fields which represent the pointers to the first and last literal of its occurrence list (labels "ofirst" and "olast", pointers have a black dot at the tail). Similar to the implementation of child lists, these pointers are one part of the occurrence list implementation. The second part is formed by the dotted pointers from one occurrence of a variable to the next, like for the two positive occurrences of variable a in the picture (labels "onext" and

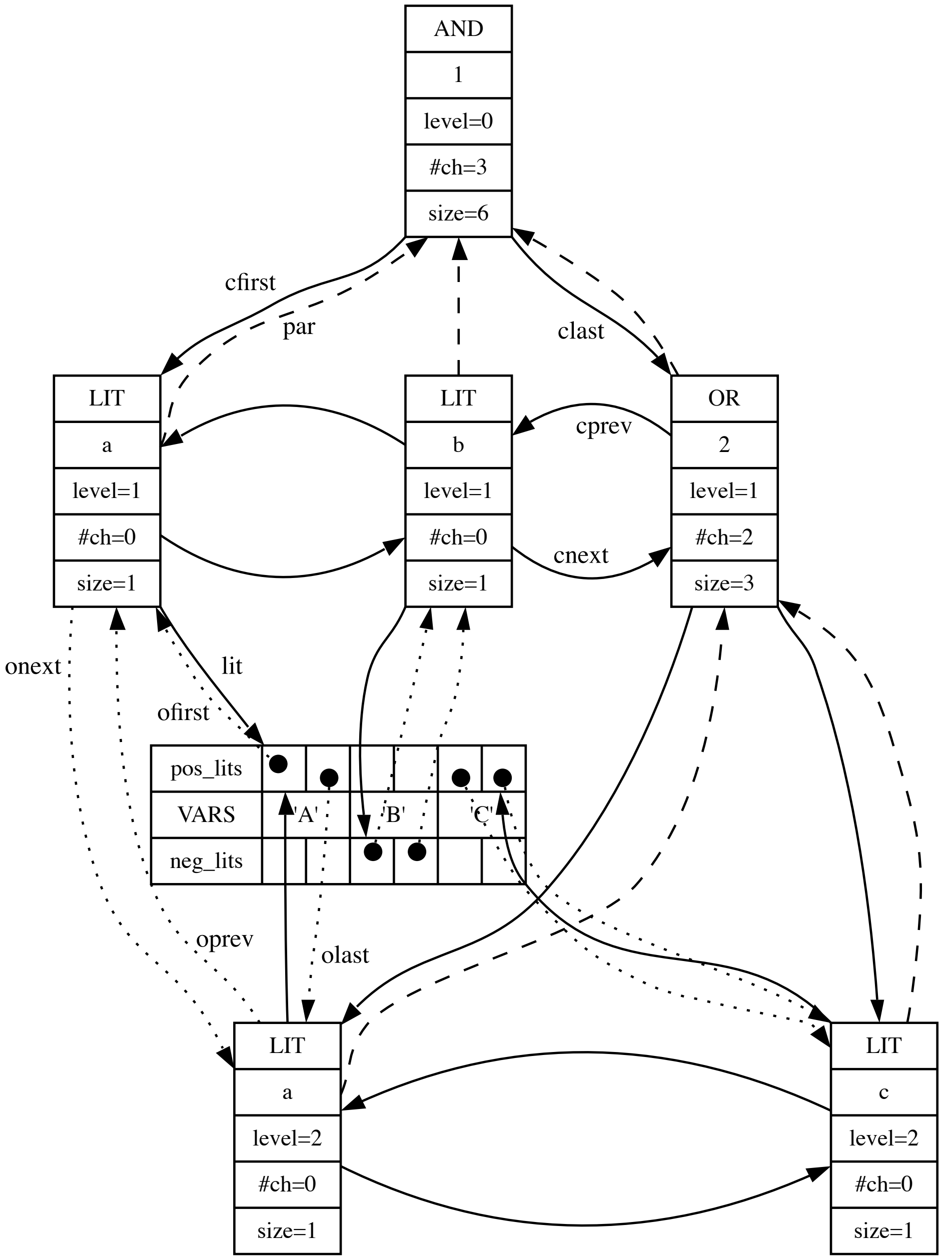


Figure 3.8: Low-level pointers in tree representation of formula $(a \wedge \neg b \wedge (a \vee c))$

"oprev"). Note that the positive literal of variable a is the only one which has more than one occurrence, hence all other literal nodes (b, c) in the graph do not have occurrence links. For example, the negative literal of variable b has no occurrence links and it is the first and last entry in the occurrence list of the negative literal object of variable b.

Solid pointers which start in a literal node and which point to the literal object in the variable table are literal pointers. For negative literal nodes, these pointers point to the negative literal object, like for the negative literal of variable b in the example. By a sequence of pointer dereferences starting in a literal node, it is possible first to access the literal object, further the variable object and finally the scope object the variable belongs to.

3.4.4 Basic Graph Operations

This section introduces basic operations for maintaining an NNF-tree and its properties under certain modifications such as deleting and inserting nodes or assigning variables. Corresponding to these operations, abstract functions will be defined. Referring to child- or occurrence lists means the lists which are formed by pointers as described in section 3.4.2 (pointers are not mentioned any more).

In the following explanations, it is assumed that a given NNF-tree fulfills all structural restrictions from section 3.3. The tree may be accessed via a pointer to the root, scopes and variables are stored as described above.

Creating Nodes

New nodes may be created by calling one of two functions $new\_operator\_node(type)$ or $new\_literal\_node(var, negated)$. The former requires the parameter *type* to be either *OR-node* or *AND-node*, the latter takes a pointer to the variable and a flag whether the created literal node is to be negated or not. Both return a pointer to a new and "clean" node object.

Node Insertion, Removal and Deletion

All operations for inserting and removing nodes which are defined in this section can be carried out in constant time because both child lists and occurrence lists are implemented as doubly linked lists with pointers to the first and last entry (for doubly linked lists, see [OW02], for example). Concerning possible violations of structural restrictions which may occur after one of these operations has been carried out, there are special functions *merge_parent* and *one_level_simplify* to repair the tree structure locally (see below).

Function $add\_to\_child\_list(parent, new\_child)$ takes node new_child and appends it to the child list of $parent$ in case that new_child is an operator node. Else, if new_child is a literal node then it is prepended to the child list. This policy guarantees that literal nodes *always* occur before any operator node in the child list of a node. The importance of this convention will become apparent below in section 3.4.6 where one-level simplification is described. After a node has been added to the child list, the child counter of the parent node is incremented by one and the child's parent pointer is set to the parent. Function $unlink\_from\_child\_list(node)$ reverses the effects of $add\_to\_child\_list$.

Function $add\_to\_occ\_list(lit\_node)$ takes a literal node and appends it to the occurrence list of the respective literal object. The counter in the literal object is incremented by one. The reverse function is $unlink\_from\_occ\_list(lit\_node)$.

Function $delete\_subformula(del\_node)$ takes node del_node, unlinks it from the tree and releases all resources that were held by any of the successors of del_node. This function requires the subgraph of del_node to be traversed and hence can be carried out in linear time. Care must be taken that literal nodes are properly unlinked from their occurrence list.

Adding or removing nodes requires maintenance of node levels and subformula sizes in some situation. For the descriptions given in the following two sections, we assume that in an NNF-tree these values were set correctly for every node before any modification was made.

Maintaining Subformula Sizes

Whenever a node is inserted into or removed from the child list of its parent, the subformula sizes of the parent and of all of its predecessors have to be updated. This is achieved by function $update\_subformula\_size(parent, size)$. If $size$ is the subformula size of some node n that has been added (removed), then value $size$ has to be added to (subtracted from) the subformula sizes of all nodes p_i in $p_1, p_2, \ldots, p_{m-1}, p_m$ where $p_1 = parent(n) and p_m = root$. The path from the parent to the root is expected to be short because of structural restrictions. Maintenance of subformula sizes is one task which profits from the design decisions that have been made.

Maintaining Node Levels

Maintenance of node levels is necessary only if some node n is added to a child list and only for all nodes in the subgraph of n. This requires a traversal of this subgraph where the level of some node n_i in the subgraph is set to $level(parent(n_i))$. Function $update\_level(node)$ fulfills this task in a non-recursive, linear depth-first search traversal.

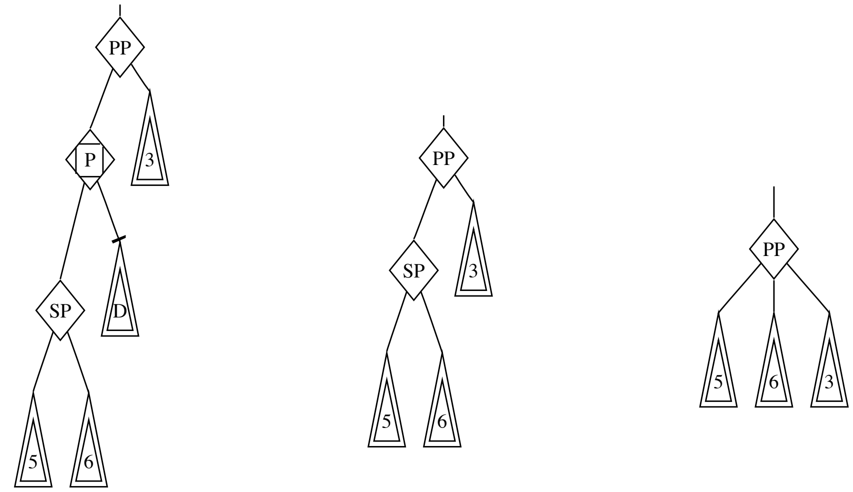


Figure 3.9: Parent merging

3.4.5 Parent Merging

It may happen that the tree violates the structural restriction of arity after a node has been deleted. This situation occurs if a parent node has only one child left after deletion of its last but one child. The leftmost tree in figure 3.9 illustrates the problem by means of a generalized tree. *P* stands for "parent", *SP* for "subparent", *PP* for "parent's parent" and *D* for "deleted". In the figure, diamond-shape nodes ◇ and ◈ may represent either an AND-node or an OR-node under the requirement of alternating types over levels. Hence if node *PP* is an AND-node then so is node *SP* and the root of subgraph *D*. Figure 3.9 illustrates function *merge_parent*(*parent*) which takes an operator node *parent* with only one child remaining and repairs the structure locally.

If subgraph *D* is deleted then node *P* will have only one child left. Since node *P* is an operator node with a minimum arity of two, it can be deleted as well which causes its remaining child (node *SP*) to be linked to node *PP* (second tree). In case that *SP* is an operator node, this situation violates the requirement of alternating types, provided that the tree had fulfilled the restriction before deletion (this is assumed). Therefore, node *SP*, which has the same type as node *PP*, is discarded too and its children are linked to node *PP* (third tree). One can think of lifting node *PP* up into node *SP* and merging both nodes, hence the name "parent merging".

In the other case, if node *SP* had been a literal node then it would have been linked to *PP* without causing any violation of alternating types (but not necessarily regarding one-level simplification – see below).

A special case in parent merging is the situation where the parent node which has only one child left is the root of the tree. In this case, no child nodes have to be moved but instead the one remaining child becomes the new root of the graph. Additionally, node levels have to be updated.

Function *merge_parent* is used to repair the structure after node deletions only, and not after inserting nodes. The structural restriction of alternating types must be preserved manually when inserting nodes by checking the types of the parent and the new child, and if the types are equal, by inserting the children of the new child rather than the new child itself.

Furthermore, *any time* literal nodes are inserted into a child list, it has to be checked whether the parent node needs to be one-level simplified. Particularly, like in figure 3.9, if *SP* was a literal node itself or had literal nodes as children, then the children of *PP* might contain double or complementary literals after linking. One-level simplification is the topic of the next section.

3.4.6 One-level Simplification

One-level simplification can remove simple redundancies in the tree by inspecting the child list of an operator node. It is necessary to remove all such redundancies, not only because operator nodes are saved but also because the implementation of advanced approaches like variable scoring (chapter 4) or redundancy removal (chapter 4) is made simpler.

When assuming that the NNF-tree is fully one-level simplified, then violations of this restriction might occur any time a literal node is added to the child list of some parent node. Thus after this operation, the parent node needs to be checked for possible one-level simplifications.

Function *one_level_simplify(parent)* works as follows if implemented in a straightforward way (a crucial refinement is presented below): the parent's child list is traversed. If a literal is encountered whose variable is not yet marked, then that variable gets positively (negatively) marked if the literal is positive (negative). Else, if the variable is positively (negatively) marked and the literal is positive (negative) then the literal may be deleted since it is redundant. In the remaining case, if the variable is positively (negatively) marked and the literal is negative (positive) then the parent is redundant and may be deleted because its child list contains two complementary literals. Finally, all marked variables have to be unmarked.

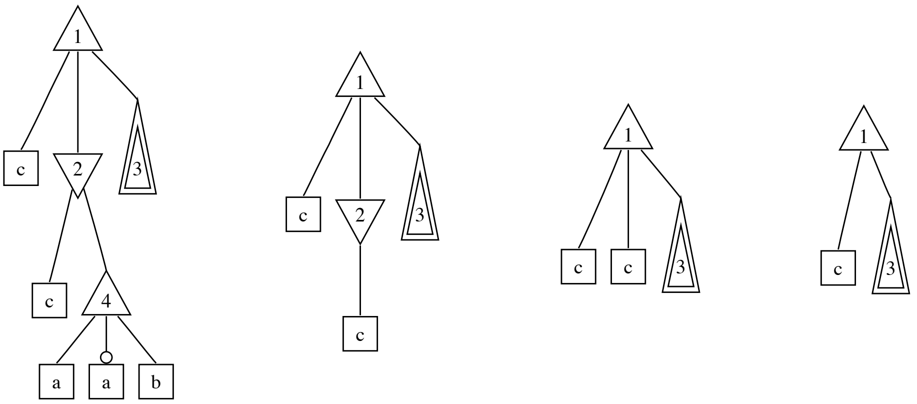


Figure 3.10: Recursive effects of one-level simplification and parent merging

The runtime of function *one_level_simplify(parent)* as presented is linear in the number of children of the parent. Test runs on real-life industrial SAT benchmarks (taken from SAT competition 2007 [SAT07]) made its drawbacks apparent: fully traversing large child lists, in this case the child lists of the root of the CNF, is too expensive. Furthermore, only literal nodes as children are relevant in one-level simplification, operator nodes not at all.

Child Lists: Literals First

In order to cope with the runtime bottleneck of one-level simplification in its straightforward implementation, a way has to be found to efficiently access the literal nodes in a child list only. In section 3.4.4 where basic functions for adding nodes to child lists have been described, an important convention with respect to this problem has been introduced: literal nodes *always* occur before any operator node in a child list.

This is a simple solution: in one-level simplification, it suffices to traverse a child list until the *first* operator node is encountered. This way, no work is wasted in searching for literal nodes any more. The runtime of function *one_level_simplify(parent)* in the revised implementation is now linear in the number of literal nodes in the child list (compared to the number of children before).

Recursive Effects

One-level simplification combined with parent merging can cause redundancies over multiple levels which are then removed recursively. Figure 3.10 shows an example. In the leftmost tree, node 4 is redundant because it contains two complementary literals of variable a. After node 4 has been deleted (second tree), node 2 has only one child left, a situation which requires parent merging. The third picture shows the tree after the remaining child of node 2 has been moved up. Since a literal node has been added to the child list of node 1, one-level simplification has to be performed. The final result is the fully one-level simplified tree on the right.

Recursive effects are not possible with one-level simplification alone because of the restriction of alternating types. For example, an AND-node which is false always has an OR-node as parent, hence it suffices to delete the AND-node. If the path from the false node up to the root consisted of adjacent AND-nodes, then all these nodes would collapse to false. Such situations can never occur.

Concerning the implementation of *delete_subformula*, *one_level_simplify* and *merge_parent*, there is indeed the possibility of indirect recursive function calls like in the previous example. Deletion could cause merging, which could cause one-level simplification which again could cause deletion and so on. It seems to be unlikely (although possible) to run into an overflow of the function call stack within such a sequence of recursive calls. Each time parent merging is carried out, the "focus" (the node which is concerned by one-level simplification) moves up towards the root. Because of the structural restrictions, node levels are expected to be small and so the number of possible recursive calls. In fact, this is the only recursive part in the implementation of Nenofex.

3.4.7 Assigning Variables

The nature of variable assignments in Nenofex is different from that in a DPLL-like search-based solver. In the latter, it must be possible to reverse the effects that have been produced by assigning a decision variable during backtracking: in order to be able to try out both values of a variable, the original formula must be recovered.

This is different if variables in a formula in NNF are eliminated by expansion. The two possible assignments of a variable and its effects are encoded in the formula simultaneously at the cost of formula size. When a variable is expanded, then subformulae which contain literals of that variable will either become reduced in size or collapse to a constant truth value, depending on the value of the variable and the polarity of the literal. Therefore, nodes which correspond to reduced parts of such subformulae are immediately deleted from the tree.

Assigning variables and propagating the effects is realized in functions *assign_variable(var, value)*, *propagate_truth*(*literal*) and *propagate_falsity(literal)*. Function *assign_variable(var, value)* takes a variable and the value to be assigned. First, the list of negative occurrences is traversed. If value is **true** (**false**) then function *propagate_falsity(literal)* (*propagate_truth(literal)*) is called on every negative literal, and vice versa for all positive occurrences. Function *propagate_truth(literal)* takes a literal and propagates the effects that are caused if the literal becomes **true**. The effects depend on the parent node. If the parent is an OR-node, then it is deleted since its boolean function is constant **true**. Else if it is an AND-node, then the literal is deleted. Function *propagate_falsity(literal)* works analogously.

Note that because of structural restrictions, the effects of propagating variable assignments occur locally at the parent nodes of the respective literals. For example, there are never two or more adjacent AND-nodes (OR-nodes) on the path from a literal up to the root. Hence propagating truth values will affect the literal itself or its parent, but never any node farther away from the literal in direction towards the root. Only parent merging and one-level simplification may cause additional deletions. Assigning variables as described is applied during the elimination of units and unates.

3.4.8 Eliminating Units

The notion of units has been introduced for formulae in CNF in section 2.2.3 (page 12). In case of an NNF-tree, a literal is unit if its parent is an AND-node and is the root of the tree. Units can be detected efficiently when relying on the convention from section 3.4.4 that literal children occur before any operator children in a child list. Unit elimination works the same way as on CNF as far as assigning the variable which a literal belongs to is concerned. If the tree root is an AND-node, function *simplify_eliminate_units* successively accesses the first child of the tree root and assigns the respective variable if the child is a literal in order to eliminate the unit. Further variables may become unit which happens whenever one of their literals is moved up to the root during parent merging. Unit elimination will run until saturation and ends if the first child of the root is an operator node or the whole tree has been deleted.

3.4.9 Eliminating Unates

A variable becomes unate if it has either only positive literals or only negative literals left. Since a counter is maintained whenever literals are added to or removed from occurrence lists of variables, it suffices to catch the situation when a counter decreases from one down to zero in order to detect unates efficiently. Different from unit elimination, where no auxiliary data structure is used for storing units, unate variables are collected in a separate list. Function *simplify_eliminate_unates* successively assigns variables from this list as described in section 3.4.7. Any variable which becomes unate during elimination will be added to the list. As with unit elimination, this process runs until saturation.

Chapter 4

Expansion

The topic of this chapter is expansion for NNF, the core function in Nenofex. In two steps, an expansion method is developed which copies the relevant parts of a formula only.

The first step (section 4.1) is an observation: applying expansion strictly according to the formal definitions in section 2.2.4 (page 13) will likely yield an expanded formula which contains parts that have been copied unnecessarily. It is shown that the expanded formula can be postprocessed until all redundancies, which correspond to unnecessarily copied parts (and only those), have been removed. Redundancy which has been present before expansion or which has been added by copying relevant parts of the formula is ignored in this chapter. Postprocessing is done by applying distributivity of conjunction and disjunction.

In the second step (section 4.2), the effects of postprocessing are anticipated in a revised expansion method called *local expansion* which, for some variable to be expanded, always yields a formula which is free of unnecessarily copied parts (but not free of redundancy in general). We conjecture that the resulting, expanded formula is minimal with respect to size increase. Our method is closely related to the CNF-based approaches for universal expansion in [Bie04] and [BB07], and to the technique of miniscoping [AB02].

Finally, in section 4.3 the implementation of local expansion is described, where emphasis is put on algorithms for identifying relevant parts of a formula. A special case of expansion is mentioned where the size of the formula does not increase.

4.1 Full Expansion and Postprocessing

Expansion of some variable x in formula F will almost double the size of the formula if carried out in a straighforward way, called *full expansion*, as

$$F \equiv F[x/0] \vee F[x/1] \tag{4.1}$$

The size of the resulting formula may be reduced in a postprocessing step where common subformulae, which have not been affected during expansion, are factored out by applying distributivity. Certain (not all) subformulae which contain occurrences of x will be affected, since variable x is set to true and false in the respective copies of the formula.

In the following example (omitting the prefix)

$$F \equiv A \wedge (B \vee (C \wedge X_1 \wedge X_2)) \tag{4.2}$$

formula F in NNF contains subformulae A, B and C which do not contain variable x, and subformulae X_1 and X_2 which contain all occurrences of x. Assume that all variables are existentially quantified. Variable x can be expanded as defined above:

$$\begin{aligned} F \equiv\ & (A \wedge (B \vee (C \wedge X_1 \wedge X_2)))[x/0] \vee \\ & (A \wedge (B \vee (C \wedge X_1 \wedge X_2)))[x/1] \end{aligned} \tag{4.3}$$

Subformulae A, B and C will not change when setting x to true or false. Thus the scope of the restriction operators [x/0] and [x/1] may be reduced:

$$\begin{aligned} F \equiv\ & (A \wedge (B \vee (C \wedge X_1 \wedge X_2))[x/0]) \vee \\ & (A \wedge (B \vee (C \wedge X_1 \wedge X_2))[x/1]) \end{aligned} \tag{4.4}$$

$$\begin{aligned} F \equiv\ & (A \wedge (B \vee (C \wedge X_1 \wedge X_2)[x/0])) \vee \\ & (A \wedge (B \vee (C \wedge X_1 \wedge X_2)[x/1])) \end{aligned} \tag{4.5}$$

$$\begin{aligned} F \equiv\ & (A \wedge (B \vee (C \wedge (X_1 \wedge X_2)[x/0]))) \vee \\ & (A \wedge (B \vee (C \wedge (X_1 \wedge X_2)[x/1]))) \end{aligned} \tag{4.6}$$

In formulae 4.3 to 4.6, expressions $[x/0]$ and $[x/1]$ have been successively moved inside to subformula $(X_1 \wedge X_2)$, which contains all occurrences of x. A, B and C have been copied unnecessarily and can be factored out:

$$\begin{aligned} F \equiv\ & (A \wedge (B \vee (C \wedge (X_1 \wedge X_2)[x/0]))) \vee \\ & (A \wedge (B \vee (C \wedge (X_1 \wedge X_2)[x/1]))) \end{aligned} \tag{4.7}$$

$$\begin{aligned} F \equiv\ & A \wedge (\ (B \vee (C \wedge (X_1 \wedge X_2)[x/0])) \vee \\ & (B \vee (C \wedge (X_1 \wedge X_2)[x/1]))\ ) \end{aligned} \tag{4.8}$$

$$\begin{aligned} F \equiv\ & A \wedge (\ B \vee (C \wedge (X_1 \wedge X_2)[x/0]) \vee \\ & B \vee (C \wedge (X_1 \wedge X_2)[x/1])\ ) \end{aligned} \tag{4.9}$$

$$F \equiv A \wedge (\ B \vee (C \wedge (X_1 \wedge X_2)[x/0]) \vee (C \wedge (X_1 \wedge X_2)[x/1])\ ) \tag{4.10}$$

$$F \equiv A \wedge (\ B \vee (C \wedge (\ (X_1 \wedge X_2)[x/0] \vee (X_1 \wedge X_2)[x/1]\ ))\ ) \tag{4.11}$$

In 4.8, A has been factored out. The copy of B can be deleted in 4.9 which was obtained from 4.8 by applying associativity of disjunction. Finally, C can be factored out in 4.10, yielding 4.11 which is the minimal formula that can be obtained by expanding x. Redundancy which is possibly added by copying affected parts of the formula (like X_1 and X_2 in the example) is ignored with this respect.

Generally, for some formula and a variable to be expanded, the *minimal expanded formula* is defined to be the expanded formula which is free of unnecessarily copied parts.

Factoring out common unaffected subformulae in a fully expanded formula like in equation 4.1 or 4.7 can reduce the size of the formula considerably. It is possible to obtain the minimal expanded formula with respect to some variable x by postprocessing as described.

However, if expansion was implemented this way, then this method would turn out to be impractical in many cases. First, for a large formula, it may be problematic that the fully expanded formula fits into memory. Second, during factoring, common subformulae need to be identified efficiently, which seems to be a difficult task. Finally, copying the whole formula and discarding redundant parts afterwards is a waste of work on its own.

Therefore, it is necessary to have a method for expansion which, by constuction, yields the minimal expanded formula. Such method is called *local expansion.*

4.2 Local Expansion

In the following, first a method for local, NNF-based expansion of existential or universal variables from the innermost scope S_n, as defined in section 2.2.4 (page 13) is presented. Afterwards, this method is adapted for expansion of universal variables from scope S_{n-1}. We conjecture that these revised expansions are optimal with respect to size increase of the formula.

Local expansion can be regarded as applying miniscoping [AB02] followed by expansion: for a variable to be expanded, the scope of its quantifier is minimized by pushing the quantifier from the prefix inside the formula successively. Then the subformula in the minimized scope is expanded.

4.2.1 Innermost Expansion

Given a QBF $S_1 \dots S_n\ \phi$ in NNF with n scopes and some variable x in S_n where $\mathit{type}(S_n) = \exists$ ($\mathit{type}(S_n) = \forall$), let $\mathit{ers}(x)$ denote the *expansion-relevant subformula* of variable x, which is defined to be the smallest subformula of ϕ which contains all occurrences of x. Hence $\mathit{ers}(x) \in \mathit{subf}(\phi)$, as defined in section 2.1.5 (page 10).

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

$$S_1 \dots S_n\ \phi \equiv S_1 \dots (S_n \setminus \{x\})\ \phi[ers(x)\ /\ (ers(x)[x/0] \otimes ers(x)[x/1])] \tag{4.12}$$

where operator $\otimes = \vee$ ($\otimes = \wedge$) if $type(S_n) = \exists$ ($type(S_n) = \forall$). In rule 4.12, 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 and false, respectively. Thus expansion of x is applied *locally* and does not require factoring. Generally, factoring out is not possible any more because there are no common unaffected subformulae in formula $(ers(x)[x/0] \otimes ers(x)[x/1])$ which have been copied unnecessarily. Hence local expansion yields the minimal expanded formula.

In this respect, it is important to point out that formula $(ers(x)[x/0] \otimes ers(x)[x/1])$ may likely contain other redundancies. Coping with the general problem of redundancy removal is the topic of chapter 6. In this chapter, the term “redundancy” stands for trivial redundancies which can be avoided solely by copying the relevant parts during expansion only.

For the previous example in formula 4.2, $ers(x) = (X_1 \wedge X_2)$. Note that, for example, $(C \wedge X_1 \wedge X_2)$ is a subformula of F which contains all occurrences of x, but not the smallest one. Assuming that $x \in S_n$ and $type(S_n) = \exists$ for some prefix $S_1 \dots S_n$, local expansion of x in formula F yields (omitting the prefix)

$$F \ \equiv\ A \wedge (B \vee (C \wedge (\underbrace{(X_1 \wedge X_2)[x/0] \vee (X_1 \wedge X_2)[x/1]}_{ers(x)[x/0]\ \vee\ ers(x)[x/1]}))) \tag{4.13}$$

The result is equal to formula 4.11, which has been obtained by postprocessing.

4.2.2 Non-innermost Expansion

We consider universal expansion of variables from scope S_{n-1}, that is, the first non-innermost scope, only. Applying universal expansion strictly according to the definition in section 2.2.4 has the same drawback as pointed out above: unaffected subformulae are copied redundantly, probably together with needless duplications of existential literals.

Concerning CNF, these problems have been investigated in Quantor [Bie04], sKizzo [Ben05b] and quantifier trees [Ben05a]. 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. Then all clauses which contain occurrences of x or of any depending existential variable are copied during expansion. This idea is generalized in [BB07] to universal expansion from arbitrary scopes. In the following, we take the definitions and notation from [BB07] and adapt them for NNF.

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

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

Let $\mathit{urs}(x, D_x)$ denote the *expansion-relevant subformula* of universal variable x with respect to D_x, which is defined to be the smallest subformula of ϕ which contains all occurrences of x and all occurrences of any existential variable $y \in D_x$. We define *local expansion* of variable x in ϕ as follows:

$$\begin{aligned} &S_1 \dots S_{n-1} S_n\ \phi \equiv \\ &\quad S_1 \dots (S_{n-1} \setminus \{x\})(S_n \cup D'_x)\ \phi[u\ /\ u[x/0] \wedge u'[x/1]] \end{aligned} \tag{4.14}$$

where u stands for $\mathit{urs}(x, D_x)$ and $\mathit{urs}(x, D_x)'$ is obtained from $\mathit{urs}(x, D_x)$ by substituting y' for all occurrences 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 $\mathit{urs}(x, D_x)$ is closely related to the CNF-based approaches in [Bie04] and [BB07], 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.

4.3 Implementation

This section describes the implementation of local expansion in Nenofex. First, a correspondence will be established between the concept of expansion-relevant subformulae and subtrees in the NNF-tree. The essential problem is to identify the subtree which, when copied during expansion, will yield an NNF-tree representing the minimal expanded formula. Such subtrees are called *expansion-relevant subtrees*: the expansion-relevant subformula corresponds to the expansion-relevant subtree and vice versa.

Second, algorithms for solving this problem are presented both for innermost and non-innermost expansions, where the latter include computation of the set of depending existential variables.

And third, the process of copying subtrees and assigning the expanded variable will be explained in detail (sections 4.3.3 and 4.3.4), putting the focus on the maintenance of node properties such as subformula size and level information.

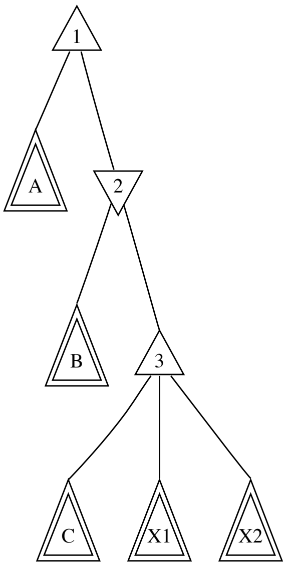


Figure 4.1: NNF-tree for $A \wedge (B \vee (C \wedge X_1 \wedge X_2))$

4.3.1 The Role of Least Common Ancestors

The notion of expansion-relevant subformulae *ers* and *urs* has been introduced above. Since every subformula corresponds to a subtree in the NNF-tree, there exists an expansion-relevant subtree in the NNF-tree which exactly represents the expansion-relevant subformula *ers* or *urs*, respectively. It suffices to unlink this subtree and replace it by an expanded subtree according to the rules of local expansion from sections 4.2.1 and 4.2.2. We describe how to find this subtree in an NNF-tree.

LCAs as an Over-approximation

Given an NNF-tree for some formula ϕ. The least common ancestor (LCA) of some variable $x \in V(\phi)$, written as $lca(x)$, is the LCA over all of its occurrences. Hence $lca(x)$ is a node which represents a subtree containing all literal nodes of variable x. Figure 4.1 shows an NNF-tree for formula 4.2 from the previous sections, where subformulae A, B and C are free of occurrences of x. Node 3 is the LCA of x. Observe that the subtree represented by node 3 does not only contain all occurrences of x, but subgraph C as well.

Node $lca(x)$ and its subtree do not necessarily correspond to the expansion-relevant subformula of x. Correctness of local expansion does not suffer if copying the subformula of $lca(x)$ instead of the expansion-relevant subformula, but the resulting, expanded formula will not be minimal in this case.

Note that, if node 3 in figure 4.1 had only $X1$ and $X2$ as children, then the subtree of $lca(x)$ would be relevant with respect to local expansion of x. Such situations are unlikely to occur in practice. For example, if formula ϕ is in CNF and some variable x has been chosen for expansion, then $lca(x)$ will be the root of the CNF (unless variable x has one occurrence only and thus is unate). Thus expanding x will copy the whole formula.

The notion of variable LCAs has to be extended in order to act as a suitable concept for identifying the expansion-relevant subtree.

Expansion-Relevant LCAs

For some variable $x \in S_n$ where $type(S_n) = \exists/\forall$ and $lca(x)$, the *expansion-relevant LCA* of x is defined by the node $lca(x)$ and all children of $lca(x)$ which contain at least one occurrence of x. Such children are called *LCA-children*.

In figure 4.1, for example, the expansion-relevant LCA of variable x comprises node 3 and its children *X1* and *X2*.

Concerning non-innermost expansion of universal variables from scope S_{n-1}, the expansion-relevant LCA is defined by the LCA over all occurrences of the expanded variable and of all depending existential variables. The definition of LCA-children is extended to children which contain at least one occurrence of x or of any depending existential variable from set D_x.

Finally, a mutual correspondence has been established between expansion-relevant LCAs, subtrees and subformulae: the expansion-relevant LCA of a variable corresponds to the expansion-relevant subtree (and vice versa), which further corresponds to the expansion-relevant subformula *ers* or *urs* (and vice versa). Expansion-relevant LCAs are the concept which allows to compute expansion-relevant subtrees in an NNF-tree.

In the following section, algorithms are presented for computing expansion-relevant LCAs and related sets of depending existential variables for non-innermost expansion.

4.3.2 Computing Expansion-relevant LCAs

The core algorithm (function *compute_lca* in algorithm 2) for computing expansion-relevant LCAs for both innermost and non-innermost expansion in Nenofex follows directly from the definition of LCAs of sets (see section 3.1 on page 15). Given two nodes, the basic idea is to carry out an explicit upward-directed search to determine the LCA. For sets of nodes, the LCA is computed incrementally by application of commutativity and associativity of the LCA operator. In order to compute LCAs

Algorithm 1: expansion_relevant_lca

```
   Input: variable var $\in V(\phi)$
   Result: expansion-relevant LCA of var as (lca, lca_children)
   Data: occurrence lists neg_occs, pos_occs of var,
         nodes lca, occ, set lca_children
1  lca $\leftarrow$ null
2  lca_children $\leftarrow \emptyset$
3  forall occ $\in$ neg_occs do
4      (lca, lca_children) $\leftarrow$ compute_lca(lca, lca_children, occ, $\emptyset$)
5  forall occ $\in$ pos_occs do
6      (lca, lca_children) $\leftarrow$ compute_lca(lca, lca_children, occ, $\emptyset$)
```

for non-innermost expansion related to subformulae urs, the algorithm is designed to work on *intermediate* expansion-relevant LCAs and their sets of collected LCA-children. In each step of incremental computation, the intermediate expansion-relevant LCA is successively enlarged.

It is important that the level of each node, which is the distance from the root, is set correctly before LCA computation. The algorithm runs in $O(1)$ space and $O(nm)$ time where n is the cardinality of the set of nodes where the LCA is computed and m is the maximum node level in the tree. The time complexity is a drawback, but the value of m is expected to be small on average due to the imposed restrictions on the tree structure.

LCAs for Innermost Expansion

The pseudo-code for computing expansion-relevant LCAs of existential or universal variables in scope S_n, which correspond to subformulae ers, is shown in algorithms 1 and 2.

Starting in algorithm 1, the pair (*lca*, *lca_children*) denotes the (intermediate) expansion-relevant LCA of variable *var*. After the data structures have been reset (line 2), the core function *compute_lca* is called on every occurrence of *var* (lines 4 and 6), where the expansion-relevant LCA is computed incrementally. Actually, function *compute_lca* takes two (intermediate) expansion-relevant LCAs (two nodes and the respective collections of LCA-children; altogether four arguments in the pseudo-code). Computation of the LCA of a variable using this function is a special case, because the second argument is an occurrence of the variable. Therefore, the respective set of LCA-children is always empty (the last argument of function *compute_lca* in lines 4 and 6).

Function *compute_lca* in algorithm 2 computes the LCA of the given intermediate LCA and an occurrence of *var* as follows. First, trivial cases are handled

Algorithm 2: compute_lca

Input: two expansion-relevant LCAs (lca1, lca_children1) and (lca2, lca_children2)
Result: (lca, lca_children), which is the (intermediate) expansion-relevant LCA of (lca1, lca_children1) and (lca2, lca_children2)
Data: nodes high, high_prev, low, low_prev, lca, collection lca_children

```
lca $\leftarrow$ null, lca_children $\leftarrow \emptyset$
if lca1 $=$ null then lca $\leftarrow$ lca2, stop
if lca2 $=$ null then lca $\leftarrow$ lca1, stop
if lca1 $=$ lca2 then
    lca $\leftarrow$ lca1
    lca_children $\leftarrow$ lca_children1 $\cup$ lca_children2
    stop
if level(lca1) $\geq$ level(lca2) then
    high $\leftarrow$ lca2, low $\leftarrow$ lca1
else
    high $\leftarrow$ lca1, low $\leftarrow$ lca2
assert(high $\neq$ low)
while high $\neq$ low do
    if level(low) $>$ level(high) then
        low_prev $\leftarrow$ low
        low $\leftarrow$ parent(low)
    else
        assert(level(high) $=$ level(low) and high $\neq$ low)
        low_prev $\leftarrow$ low
        low $\leftarrow$ parent(low)
        high_prev $\leftarrow$ high
        high $\leftarrow$ parent(high)
assert(high is the LCA of lca1 and lca2)
lca $\leftarrow$ high
if high $=$ lca1 then
    assert(lca1 is predecessor of lca2, lca2 was low and was moved upwards)
    lca_children $\leftarrow$ lca_children1 $\cup$ {low_prev}
else if high $=$ lca2 then
    assert(lca2 is predecessor of lca1, lca1 was low and was moved upwards)
    lca_children $\leftarrow$ lca_children2 $\cup$ {low_prev}
else
    assert(both high and low were moved upwards)
    lca_children $\leftarrow$ {high_prev } $\cup$ {low_prev}
```

(lines 2 and 3), where the algorithm stops immediately (these cases occur in the first call of function *compute_lca* from *expansion_relevant_lca*).

Lines 4 to 7 are relevant for LCA computation related to non-innermost expansion only (see below). The if-clause in line 4 is always false when computing the LCA of a variable from the innermost scope.

Next (line 8), the node which is closer to the root, that is higher up in the tree, is selected among *lca1* and *lca2* by level comparison.

The actual LCA computation by upward-directed search is carried out in the loop in lines 13 to 22. The loop terminates if *high* equals *low*, that is the LCA has been determined. As long as *high* and *low* have different levels, *low* is moved upwards by following parent pointers, where the previous node is remembered in each step (lines 14 to 16).

If levels of *high* and *low* are equal but the LCA has not yet been found out (line 18), then parent pointers of *high* and *low* are followed in parallel fashion, again remembering the previous node in each step (lines 18 to 22). The loop will terminate because, in line 18, both *high* and *low* have the same distance to the root of the NNF-tree (it is crucial that level information of nodes is set correctly). At the latest, this happens at the root of the NNF-tree. Node *high* is the LCA of *lca1* and *lca2* (line 24).

Finally, three cases can occur depending on the relationship between *lca1* and *lca2*. If *lca1* is a predecessor of *lca2* (line 25), then *high* (*low*) was assigned *lca1* (*lca2*) in line 11 and only *low* was moved upwards by following parent pointers until *high* was reached. Node *low_prev* is uniquely added to the set of LCA-children of *lca1* (line 27). If the subtree denoted by (*lca1*, *lca_children1*), which is the intermediate expansion-relevant LCA, did not contain node *lca2* before, then it will after node *low_prev* has been added. Analogous arguments apply for the situation where *lca2* is a predecessor of *lca1*, which is handled similarly (lines 28 to 30). If there is no successor-predecessor relationship between nodes *lca1* and *lca2* (line 31), then both *high* and *low* were moved upwards. Nodes *high_prev* and *low_prev* make up the set of LCA-children of *high* (line 33), which is the *new* intermediate LCA. The previous intermediate LCA is now contained in the subtree denoted by (*lca*, *lca_children*).

After the LCA of a some variable x has been computed, the set of LCA-children always contains at least two and at most n nodes where n is the number of children of $lca(x)$. Variables with only one occurrence are an exception, but this case should never occur in practice if unates are eliminated until saturation. Adding nodes uniquely to the set of LCA-children is implemented by marking all nodes which are members of the set and adding nodes which are not yet marked only. This operation can be carried out in constant time (sets are implemented as stacks). All marks must be cleared before the first and after the last call of

Algorithm 3: universal_expansion_relevant_lca

```
Input: variable univ_var $\in S_{n-1}$ where $type(S_{n-1}) = \forall$
Result: LCA of univ_var as (univ_lca, univ_lca_children), set $D_u$
Data: set $D_u$, nodes lca, e, set lca_children

1 $D_u \leftarrow \emptyset$
2 (univ_lca, univ_lca_children) $\leftarrow$ expansion_relevant_lca(univ_var)
3 $D_u \leftarrow D_u \cup$ collect_depending_vars(univ_var, univ_lca_children)
4 forall unprocessed e $\in D_u$ do
5     (lca, lca_children) $\leftarrow$ expansion_relevant_lca($e$)
6     (univ_lca, univ_lca_children) $\leftarrow$ compute_lca(univ_lca,univ_lca_children, lca,lca_children)
7     $D_u \leftarrow D_u \cup$ collect_depending_vars(univ_var, univ_lca_children)
```

Algorithm 4: collect_depending_existentials

```
Input: var $\in S_{n-1}$ where $type(S_{n-1}) = \forall$, set univ_lca_children
Result: set D
Data: set D, nodes u, s

1 D $\leftarrow \emptyset$
2 forall unmarked u $\in$ univ_lca_children do
3     mark(u)
4     forall unmarked s $\in$ Successors(u) do
5         mark(s)
6         if is_literal_node(s) and var(s) $\in S_n$ then
7             D $\leftarrow$ D $\cup$ {var(s)}
```

compute_lca from algorithm 1. The time required for clearing marks is linear in the cardinality of the set.

LCAs for Non-innermost Expansion

Given a universal variable *univ_var* $\in S_{n-1}$, the result of algorithm 3 is the expansion-relevant LCA (corresponding to subformula *urs*), denoted by (*univ_lca*, *univ_lca_children*), and the set D_u of depending existential variables, both of which are computed incrementally.

First, the expansion-relevant LCA of *univ_var* is computed according to algorithm 1. The result is the intermediate LCA and LCA-children which contain all occcurrences of the universal variable (line 2). Next, all existential variables from scope S_n which have occurrences in the subtree denoted by (*univ_lca*, *univ_lca_children*) are collected (line 3).

Collecting depending variables is carried out in algorithm 4 by traversing the subtree denoted by (*univ_lca*, *univ_lca_children*). With this respect, it is important not to visit the whole subtree with root *univ_lca* since this node might have children whose subtrees do not contain occurrences of *univ_var*. Thus only the subtrees of LCA-children which have been collected so far need to be considered, and further, only those which have not yet been traversed. This can be achieved by marking LCA-children whose subtree is traversed (line 3) and ignoring those which are marked (line 2). For each unmarked LCA-child, all unmarked successors are visited (line 4). The need for marking successors at this point (line 5) will become apparent below: the subtree denoted by (*univ_lca*, *univ_lca_children*) is enlarged similar to algorithm 1 and it must be avoided to visit nodes more than once. If a successor of an LCA-child is a literal node of some existential variable in scope S_n, then the respective variable is added to set D. After the outer loop has terminated, set D contains all depending existential variables which have occurrences in the subtree of the intermediate expansion-relevant LCA of *univ_var*. At the end, marks of visited LCA-children or nodes are *not* cleared.

In algorithm 3, the intermediate expansion-relevant LCA of *univ_var* is enlarged by taking the LCAs of all depending existential variables into account (lines 5 to 7).

For each depending variable in set D_u which has not been processed (line 4), first the expansion-relevant LCA denoted by (*lca*, *lca_children*) is computed (line 5) as described in algorithm 1.

Enlarging the intermediate LCA of *univ_var* (line 6) is done by means of function *compute_lca* in algorithm 2, which is called on (*univ_lca*, *univ_lca_children*), the intermediate LCA of the universal variable and on (*lca*, *lca_children*), the expansion-relevant LCA of a depending variable.

In algorithm 2, trivial cases in lines 2 and 3 can not occur when computing LCAs for non-innermost expansion. Apart from that, figures 4.2 to 4.5 illustrate all possible relationships between *lca1* (in this case the universal LCA) and *lca2* (the LCA of the depending existential variable) to be handled in algorithm 2. In the figures, diamond-shape nodes represent operator nodes of arbitrary type, nodes marked with black dots are LCA-children and nodes labelled R denote arbitrary subgraphs.

First, if *lca1* and *lca2* are equal (line 4), then the respective sets of LCA-children are unified and the algorithm terminates. In figure 4.2, LCA-children of *lca1* and *lca2* are marked with black triangles and dots.

Otherwise, LCA computation is carried out as described in the previous section in lines 8 to 24. For the cases in lines 25 and 28, figure 4.3 (figure 4.4) illustrates the situation where *high* is a predecessor of *low* and where *low_prev* is already contained (not contained) in the set of LCA-children of *high*. Dashed lines in the figures indicate paths of arbitrary length which are traversed by following parent pointers in lines 14 to 16. Figure 4.5 shows the case where both *high* and *low* need to be moved upwards. Node L is the LCA of the two. Dotted lines are traversed

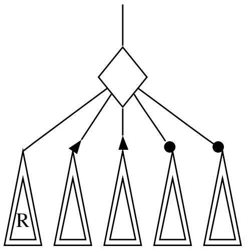


Figure 4.2:

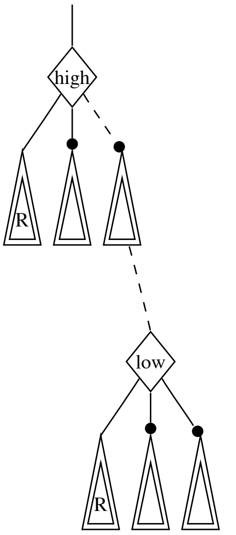


Figure 4.3:

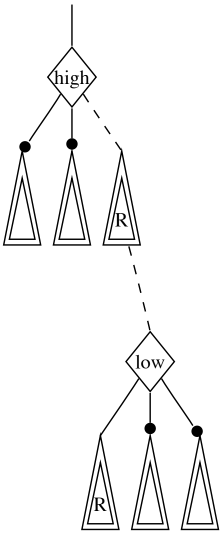


Figure 4.4:

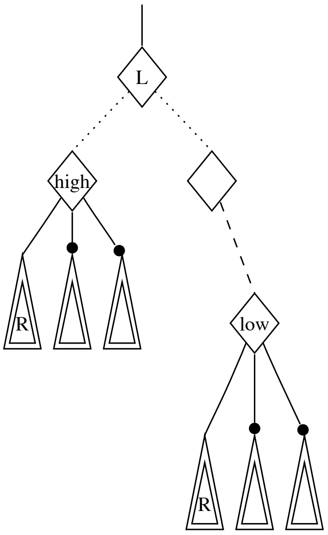


Figure 4.5:

by following parent pointers in parallel fashion in lines 17 to 22.

After function *compute_lca* has been called on the intermediate universal LCA and the expansion-relevant LCA of the depending existential variable (line 6 in algorithm 3), the parts of the (possibly) enlarged subtree denoted by (*univ_lca*, *univ_lca_children*) which have not been visited are traversed and depending variables not yet collected are added to set D_u (line 7). The loop in line 4 ends if all variables from the (possibly) growing set D_u have been processed.

The purpose of marking nodes and variables is threefold. First, it is needed for adding LCA-children in algorithm 2. Second, for marking visited nodes in algorithm 4 in lines 3 and 5. It is crucial that each node in the subtree denoted by (*univ_lca*, *univ_lca_children*) is visited exactly once during the whole process of incremental LCA computation and variable collection. For this kind of marks, an optimized marking policy has been implemented which differs from the pseudo code: it suffices to mark visited LCA-children only, in contrast to marking all nodes in the subtree, and ignoring all successors of visited nodes in forthcoming graph traversals. This reduces the effort of clearing marks afterwards. The third class of marks are variable marks: any depending existential variable which has been added to set D_u is marked, which allows adding variables in constant time similar to adding LCA-children. All marks of whatever kind must be cleared before and at the end of algorithm 3.

4.3.3 Innermost Expansion

This section deals with maintenance and modifications of an NNF-tree during local expansion of a variable from scope S_n, where $\mathit{type}(S_n) = \exists/\forall$. Any time the tree is modified, node properties such as level information or subformula size must be updated. We make the following assumptions:

- the expanded variable has more than one occurrence
- the set of LCA-children does not contain a literal of the expanded variable

Concerning the first assumption, the situation will never occur if unates are eliminated until saturation (which is expected). The second assumption is related to a special case of local expansion which does not require copying subtrees (see section 4.3.5 below).

For some variable $x \in S_n$ and its expansion-relevant LCA (*lca*, *LCA-children*) as computed by algorithm 1, the NNF-tree has to be transformed according to the rule of local expansion as defined in equation 4.12. Depending on the type of the expanded variable, the type of node *lca* and the cardinality of set *LCA-children* compared to the number of children of *lca*, eight situations can occur which require different operations with respect to copying subtrees and maintaining node properties.

We introduce a symbolic notation: a triple $\langle \mathit{var}, \mathit{lca}, \mathit{cardinality} \rangle$ denotes one of the eight possible cases where $\mathit{var} \in \{\forall, \exists\}$ represents the type of the variable, $\mathit{lca} \in \{\vee, \wedge\}$ the type of the LCA and $\mathit{cardinality} \in \{=, <\}$ the relation between the set of collected LCA-children and the set of children of the LCA. Symbol $<$ represents the situation where the set of LCA-children is a proper subset of the set of children of *lca*, otherwise symbol $=$ is used if the two sets are equal.

This notation will be applied both for innermost and non-innermost expansion as in the next section. For example, the triple $\langle \exists, \wedge, = \rangle$ denotes the case of expanding an existential variable which has an AND-node as LCA and the number of collected LCA-children equals the number of children of its LCA. The situation in figure 4.1 can be described by $\langle \exists, \wedge, < \rangle$. Note that the set of collected LCA-children always contains at least two nodes and at most as many as the number of children of the respective LCA in the pair (*lca*, *LCA-children*).

Figures 4.6 to 4.9 show parts of an NNF-tree before (left) and after (right) local expansion of an existential variable has been carried out, and hence can be regarded as transformation templates. A node labelled L stands for the LCA or its copy, as in figure 4.8, edges marked with black dots indicate LCA-children or their copies and nodes labelled R are the remaining children of the LCA which do not contain occurrences of the expanded variable. Nodes which are labelled [1] ([0]) correspond to subtrees where the expanded variable is set to true (false).

This kind of label represents the restriction operators $[x/1]$ and $[x/0]$ as applied in the rule of local expansion (equation 4.12).

Concerning necessary tree modifications, only the four cases $\langle \exists, *, * \rangle$ will be described in detail because each of these cases has a dual one (written as "$\approx$") in $\langle \forall, *, * \rangle$. Dual cases can be treated by applying the same sequences of transformations, provided that the types of nodes are adapted to universal expansion (see below):

$$\begin{array}{ccc}
\langle \exists, \vee, = \rangle & \approx & \langle \forall, \wedge, = \rangle \\
\langle \exists, \vee, < \rangle & \approx & \langle \forall, \wedge, < \rangle \\
\langle \exists, \wedge, = \rangle & \approx & \langle \forall, \vee, = \rangle \\
\langle \exists, \wedge, < \rangle & \approx & \langle \forall, \vee, < \rangle
\end{array}$$

Figures 4.6 to 4.9 represent the dual universal cases as well if node types are inverted (AND-node instead of OR-node and vice versa; corresponding figures A.1 to A.4 may be found in the appendix).

Copying subtrees is performed in a non-recursive, linear depth-first search traversal where nodes are copied in postorder. Any node properties, in particular level information, are copied as well. In most situations (see below), this can save setting node levels in the copied subtree in an additional traversal after copying.

In the forthcoming descriptions, it is assumed that the basic graph operations are used for linking and unlinking nodes or maintaing node properties (see section 3.4.4 on page 30). In the implementation, tree modifications during expansion never cause violations of structural restrictions. All restrictions are fulfilled afterwards.

Cases $\langle \exists, \vee, = \rangle$ and $\langle \exists, \vee, < \rangle$

The rule of local expansion requires that the expansion-relevant subformula $ers(x)$ is replaced by $ers(x)[x/0] \vee ers(x)[x/1]$, which corresponds to replacing the expansion-relevant subtree denoted by (*lca*, *LCA-children*) by a disjunction over two copies of the subtree (actually, the original subtree and one copy). Since the LCA, which is the root of the subtree, is an OR-node as well, the structural restriction of alternating types over levels forces the adjacent OR-nodes to be merged into one node. This has the same effect as if copies of all collected LCA-children were added to the LCA in advance. The following sequence of operations has to be carried out:

- for each LCA-child *lca-ch* in *LCA-children*, generate a copy *lca-ch-copy* and add it to the child list of node *lca* (node L in figures 4.6 and 4.7).

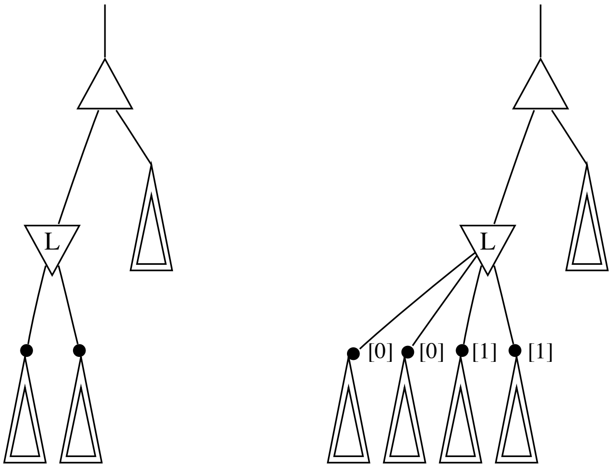


Figure 4.6: Expansion template for case $\langle \exists, \vee, = \rangle$

- update the subformula sizes of node *lca* and of all its predecessors by adding the sum over the subformula sizes of all nodes *lca-ch-copy* that have been generated in the previous step

Figures 4.6 for case $\langle \exists, \vee, = \rangle$ and 4.7 for case $\langle \exists, \vee, < \rangle$ show the the NNF-trees before and after expansion. Node L is an OR-node. Note that it is not necessary to explicitly set level information in the subtree of a node *lca-ch-copy* because levels have been copied and the original LCA-child and its copy have the same parent after expansion. Thus node levels are correctly set after copying. The transformations can be applied to the dual cases $\langle \forall, \wedge, = \rangle$ and $\langle \forall, \wedge, < \rangle$ without any modification.

Case $\langle \exists, \wedge, = \rangle$

Like in the previous case, local expansion requires the subtree of (*lca*, *LCA-children*) to be replace by a disjunction over two copies of the subtree. Since the parent of *lca* is an OR-node, replacing the subtree will insert an OR-node into the child list of the parent of *lca*, thus violating the structural restriction of alternating types: the adjacent OR-nodes must be merged. As before, this effect can be anticipated by applying the following sequence of operations:

- generate a copy *lca-copy* of the whole subtree denoted by node *lca*

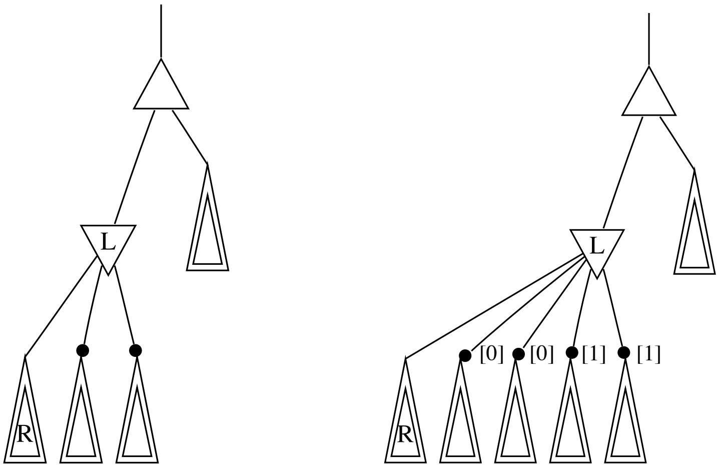


Figure 4.7: Expansion template for case $\langle \exists, \vee, < \rangle$

- add *lca-copy* to the child list of the parent of *lca* (*lca* and *lca-copy* will be siblings then)
- update the subformula sizes of the parent of *lca* and of all its predecessors by adding the subformula size of *lca-copy*

Again, levels are correctly set during copying the subtree, no additional traversal of the copy is necessary. Figure 4.8 illustrates the transformation, where node L is an AND-node. Like before, the transformations can be applied to the dual case $\langle \forall, \vee, = \rangle$ without any modification.

There is a special case (not shown in the figures) if *lca* is the root of the NNF-tree, that is *lca* has no parent. A new tree root has to be generated during expansion:

- create a new OR-node *new-root*
- make *new-root* the root of the tree
- add *lca* to the child list of *new-root*
- update level information in the subtree of *lca* by traversal (for all nodes, levels will be increased by one)
- generate a copy *lca-copy* of the whole subtree denoted by node *lca*

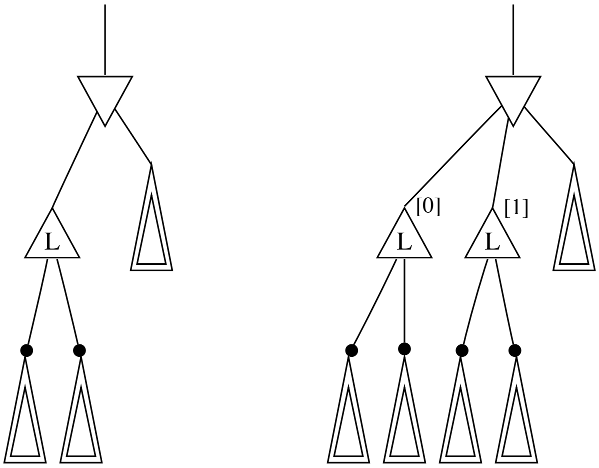


Figure 4.8: Expansion template for case $\langle\exists, \wedge, =\rangle$

- add *lca-copy* to the child list of *new-root*
- set the subformula size of *new-root* to $1 + 2 * size(lca)$

Level information which has been set in the subtree of *lca* will be copied when generating *lca-copy*. In order to handle this special situation in the dual case $\langle\forall, \vee, =\rangle$, AND-nodes have to be created instead of OR-nodes.

Case $\langle\exists, \wedge, <\rangle$

Different from the previous cases, inserting the disjunction over the two copies of the subtree of (*lca*, *LCA-children*) does not violate the structural restriction of alternating types. Since *lca* is an AND-node and the set of LCA-children a proper subset of its children (there is at least one child which does not contain an occurrence of the expanded variable), the disjunction has to be added to the child list of *lca* and the copies of the subtree denoted by (*lca*, *LCA-children*), which have an AND-node as root, have to be linked to the disjunction.

This situation is shown in figure 4.9. Subtree R does not contain an occurrence of the expanded variable. The types of L, which represents the LCA, and S are AND-node and OR-node, respectively. Node S, a “split node”, is the disjunction over two copies of the expansion-relevant subtree and can be regarded to semantically “split” the expanded subtree with root S into two branches where the expanded variable is set to true and false. In the previous expansion cases, the

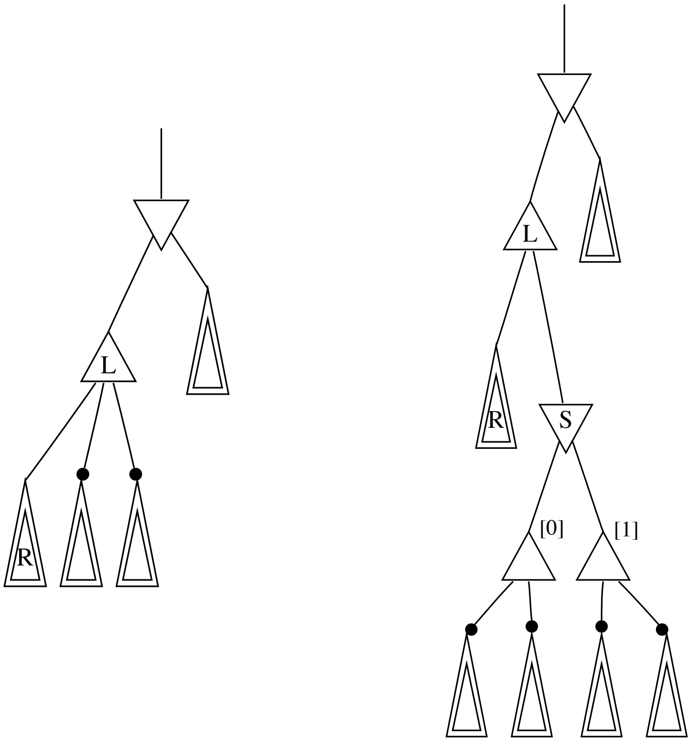


Figure 4.9: Expansion template for case $\langle \exists, \wedge, < \rangle$

split node is implicitly represented by the (already present) OR-node where the copies of the expansion-relevant subtrees have been inserted.

The described expansion is performed in the following sequence of operations:

- create a new OR-node *split-or*
- add *split-or* to the child list of *lca*
- create a new AND-node *new-and* and set its subformula size to one
- add *new-and* to the child list of *split-or* (in figure 4.9, *new-and* will be a child of S then)

- for each LCA-child *lca-ch* in *LCA-children*, unlink *lca-ch* from *lca*, add it to the child list of *new-and* and add the subformula size of *lca-ch* to the one of *new-and*. The subtree with root *new-and* now represents the expansion-relevant subtree.

- update level information in the subtree of *new-and* by traversal (for all nodes, levels will be increased by two)

- generate a copy *new-and-copy* of the whole subtree denoted by node *new-and*

- add *new-and-copy* to the child list of *split-or*

- set the subformula size of *split-or* to $1 + 2 * \mathit{size}(\mathit{new\text{-}and})$

- update the subformula sizes of the parent of *split-or* and of all its predecessors by adding $1 + 1 + \mathit{size}(\mathit{new\text{-}and})$: one for *split-or*, one for *new-and-copy* and the size of the subgraph of *new-and*

Apart from the special case in $\langle \exists, \wedge, = \rangle$ where the LCA is the root of the tree, this is the only situation where levels are set explicitly by traversing the subtree. Concerning the dual case $\langle \forall, \vee, < \rangle$, AND-nodes have to be created instead of OR-nodes and vice versa.

Figures 4.10 to 4.12 show expansion of variable x in formula

$$(a \vee b) \wedge (\neg x \vee c) \wedge (\neg x \vee d) \wedge (x \vee e) \wedge (x \vee f)$$

which is in CNF. In figure 4.10, the root is the LCA of variable x and nodes $C1$ to $C4$ represent clauses containing variable x and hence form the set of LCA-children. The expansion-relevant formula is $C1 \wedge C2 \wedge C3 \wedge C4$. Figure 4.11 shows the graph after relevant subtrees have been copied according to case $\langle \exists, \wedge, < \rangle$, but before x has been assigned and propagated. Labels $[x/0]$ and $[x/1]$ indicate the parts where x will be set to true and false. The expanded formula after assigning the variable is shown in figure 4.12. Note that parent merging causes the literals to move up to the two conjunctions.

Assigning the Expanded Variable

After the different cases in expansion have been treated as described above, all node properties are set correctly. It remains to set the expanded variable to true and false in the respective parts of the NNF-tree. By convention, the variable is set to false in the original subtrees (labelled "[0]" in the figures) and to true in the copied ones (labelled "[1]" in the figures). Assigning the variable is carried out as described in section 3.4.7 (page 35).

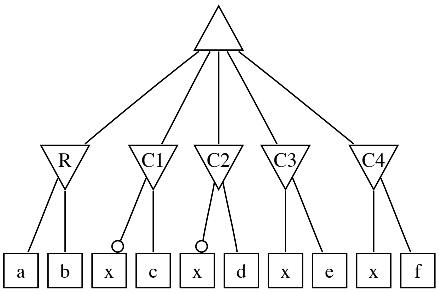


Figure 4.10: Example for case $\langle \exists, \wedge, < \rangle$ – original formula

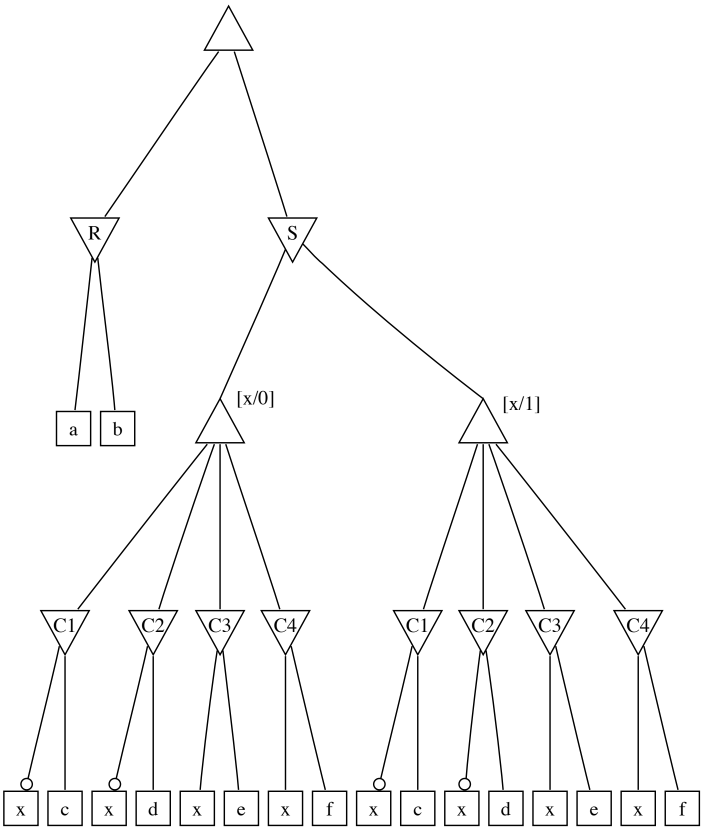


Figure 4.11: Example for case $\langle \exists, \wedge, < \rangle$ – before propagating variable assignments

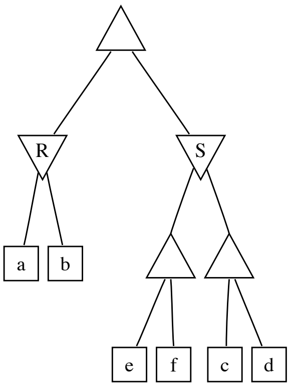


Figure 4.12: Example for case $\langle\exists, \wedge, <\rangle$ – the expanded formula

Since occurrence lists are traversed during assigning a variable, a way has to be found to distinguish occurrences in the original subtrees from the ones in the copied subtrees. This is achieved by marking all copied occurrences of the expanded variable during copying (rather than in an additional traversal of the copied subtrees). For all marked occurrences the variable will be set to true, and to false otherwise.

A Possible Optimization in Level Maintenance

Case $\langle\exists, \wedge, <\rangle$ and the special situation in $\langle\exists, \wedge, =\rangle$ and the dual cases for universal expansion require node levels to be set explicitly in an additional traversal of the affected subtree. In all other cases, level information is copied.

In an advanced approach, which has not been implemented, node levels could be set directly during copying, both in the original subtree and in the copied one, because the difference between the old and new value of a node's level (before and after expansion) is known in advance. Thus local expansion can be carried out by visiting nodes in the expansion-relevant subtree exactly once during copying, where levels, subformula sizes and occurrence marks for assigning the variable are set in one pass. In addition, the subformula sizes of all predecessors of the "split-node" (the node where copies of subtrees are linked to) must be updated, which is not expected to be expensive due to the structural restrictions for keeping node levels low.

4.3.4 Non-innermost Expansion

Expansion of universal variables from scope S_{n-1} can be described by taking into account only the four cases $\langle\forall, *, *\rangle$ among the eight possible cases. The same two assumptions are made as in the previous section at the beginning. The expansion-relevant LCA, denoted by $urs(x, D_x)$, is computed by algorithm 3. For the respective cases $\langle \forall, *, * \rangle$, exactly the same sequences of operations are applied as listed above for the dual cases $\langle \exists, *, * \rangle$. Additionally, depending existential variables from set D_x are duplicated and added to the innermost scope before subtree copying. Occurrences of any variable in D_x in the original subtree will be replaced by occurrences of the duplicated variable in D'_x in the copied subtree. During subtree copying, checking if an existential variable occurs in set D_x, and hence has been duplicated, can be done in constant time if a pointer is set from each original variable in D_x to its duplicate when variables are copied.

4.3.5 A Special Case: Non-increasing Expansions

In the previous two sections, the assumption has been made that the set of LCA-children does not contain a literal of the expanded variable, hence the expansion-relevant LCA of some variable x does not have a literal of x as a child. In the opposite case, the expanded variable can immediately be assigned a value (see below) without copying subtrees. The reason is that copied subtrees would be deleted by assigning the expanded variable if ordinary, local expansion had been carried out as described. These effects can be anticipated by assigning the expanded variable in advance, which causes the formula to decrease in size. This kind of "non-increasing" expansions are applicable both for innermost and non-innermost expansions, in any of the eight cases defined above. Non-increasing expansion corresponds to the following equivalences (see [AB02]):

$$\begin{array}{lcl@{\qquad\qquad}lcl}
\exists x\ (x \vee \phi) & \equiv & \textbf{true} & \exists x\ (x \wedge \phi) & \equiv & \phi[x/1] \\
\exists x\ (\neg x \vee \phi) & \equiv & \textbf{true} & \exists x\ (\neg x \wedge \phi) & \equiv & \phi[x/0] \\
\forall x\ (x \wedge \phi) & \equiv & \textbf{false} & \forall x\ (x \vee \phi) & \equiv & \phi[x/0] \\
\forall x\ (\neg x \wedge \phi) & \equiv & \textbf{false} & \forall x\ (\neg x \vee \phi) & \equiv & \phi[x/1]
\end{array}$$

In the rules, x and $\neg x$ can be regarded as the literal in the set of LCA-children and the boolean operators as the LCA of some variable x.

The value that is assigned to the "expanded" variable depends on its type and on the polarity of the literal. If the variable is existential and the literal positive (negative), the variable can be set to true (false). Else, if the variable is universal and the literal positive (negative), the variable is assigned false (true).

Chapter 5

Variable Scores

Eliminating variables by expansion or resolution as in Nenofex or Quantor [Bie04] may result in an increase of the formula size which could yield the elimination process impractical. A similar situation applies for search-based, exponential-time decision procedures for SAT [SAT07]. The observed efficiency and feasibility of these procedures relies on certain structural properties of problem instances in practice, where approaches like learning or backjumping [SS96] can benefit from. Concerning structured QBF, variables likely differ considerably in the amount of size increase that is caused if the variable is eliminated by expansion or resolution. A heuristic approach for selecting the variable to be eliminated can profit from this property. For example, in Quantor the scheduling of variable eliminations is based on a cost measure which considers the number of literals that are added to the formula in an elimination step.

In this chapter a cost measure for expansion on NNF will be introduced which takes into account the total number of nodes in the NNF-tree before and after expanding a particular variable (section 5.1). The expansion costs of a variable are defined as the difference between the tree size after and before expansion (the size of an NNF-tree is the number of nodes in the tree). A variable is "cheaper" than another if it has lower costs, and hence would be expanded first in a greedy elimination strategy because less size increase is caused.

The *score* of a variable in Nenofex is an *upper bound* on the actual expansion costs and is used to assess variables for scheduling expansions. Scores take into account the number of nodes which are added to and removed from the NNF-tree during expansion in order to estimate the actual expansion costs as precisely as possible. Exact computation of the number of removed nodes is complex, therefore this number is an approximation when scores are computed.

Given the expansion-relevant LCA of a variable, algorithms are presented for computing its score (sections 5.2.1 and 5.2.2). Finally, a scheme for partial recomputation of scores after modifications of the NNF-tree is described (section 5.3 and 5.4). The idea is to avoid full recomputation of variable scores whenever possible by excluding variables whose score has not been affected by the recent expansion.

5.1 Definition

Given a QBF $S_1 \dots S_n\ \phi$, its NNF-tree and an existential or universal variable x from scope S_{n-1} or S_n, the *accurate expansion costs* of x are defined as

$$\mathit{cost}(x) := \mathit{size}(\phi') - \mathit{size}(\phi)$$

where formula ϕ' is obtained from ϕ by eliminating x by local expansion and *size* is the number of nodes in the NNF-tree representing the formula. This definition is refined to

$$\mathit{cost}(x) := \mathit{score}_{\mathit{inc}}(x) - \mathit{score}_{\mathit{dec}}(x)$$

where the increase score (decrease score) of x, written as $\mathit{score}_{\mathit{inc}}(x)$ ($\mathit{score}_{\mathit{dec}}(x)$), is the number of nodes which are added to (deleted from) the NNF-tree of ϕ during local expansion of variable x. Expanding a variable with positive (negative) costs will increase (decrease) the size of the formula.

Increase and decrease score are the central concepts of expansion costs in Nenofex, but accuracy of costs is traded for simplicity of computation: the *score* of a variable is defined as

$$\mathit{score}(x) := \mathit{score}_{\mathit{inc}}(x) - \mathit{score}_{\mathit{dec}}(x)$$

and is an upper bound (an over-approximation) on the accurate expansion costs, thus $\mathit{score}(x) \geq \mathit{cost}(x)$ for all variables x. In contrast to positive costs, expanding a variable with positive score could possibly decrease the size of the formula. Although the score definition is equal to the one of accurate expansion costs, scores are not exact, which has its roots in the implementation of score computation.

Given the expansion-relevant LCA of a variable, computation of increase scores can be done efficently and in a straightforward way derived from the different cases in expansion (see section 4.3.3 on page 50). Thus increase scores are accurate in Nenofex.

Concerning decrease scores, the situation is more complex. Due to the structural restrictions of the NNF-tree, node deletions can lead to further, recursive deletions (see section 3.4.6 on page 33). Although possible, recursive effects are not considered in the computation of decrease scores, which consequently are a lower bound (an under-approximation) on the actual number of deleted nodes.

The reason is that the implementation becomes complicated: all applications of parent merging and one-level simplification need to be determined in advance, which requires inspecting child lists of involved nodes recursively. Furthermore, it is questionable if exact scores are worth the effort compared to estimated, pessimistic scores in the sense of producing considerably smaller formulae in a sequence of expansions.

Since increase scores are exact, it is the computation of decrease scores in Nenofex which renders variable scores inaccurate and pessimistic.

5.2 Score Computation

Given a QBF $S_1 \dots S_n\ \phi$ and an existential or universal variable x from scope S_{n-1} or S_n, first the expansion-relevant LCA (*lca*, *LCA-children*) of x needs to be determined before its score can be computed. In general, it is assumed that variable x has more than one occurrence and that the set *LCA-children* does not contain a literal of x, hence non-increasing expansion (see section 4.3.5) can not be applied.

5.2.1 Increase Score

The increase score $score_{inc}(x)$ of a variable can be determined by counting the nodes which would be added to the NNF-tree during local expansion. This is done by anticipating the expansion case which applies to the variable as follows (*size* is the subformula size of a subtree):

- cases $\langle \exists, \vee, * \rangle$:
  $$score_{inc}(x) := \sum_{c} size(c)$$
  for all nodes c in the set *LCA-children*

- case $\langle \exists, \wedge, = \rangle$:
  $$score_{inc}(x) := size(lca) + r$$
  where $r = 1$ if $lca(x)$ is the root of the tree (a new root needs to be created) and 0 otherwise.

- case $\langle \exists, \wedge, < \rangle$:
  $$score_{inc}(x) := \sum_{c} size(c) + 3$$
  for all c in the set *LCA-children*, and three additional nodes (one disjunction and two conjunctions)

Dual cases in $\langle\forall, *, *\rangle$ are treated similarly. If the set *LCA-children* contains a literal of x then the increase score is always set to zero because non-increasing expansion can be applied.

Computing increase scores requires $O(|LCA\text{-}children|)$ time since every node in set *LCA-children* is inspected and subformula sizes are stored and maintained for each node, which allows looking up sizes in constant time.

5.2.2 Decrease Score

The decrease score $score_{dec}(x)$ of a variable can be determined by counting the nodes, and only those, which would be deleted *immediately* by assigning the variable during local expansion. Node deletions in parent merging or one-level simplification are ignored.

Variable x is *virtually* assigned true, then false, where the number of "deleted" nodes is counted each time by marking and collecting nodes which would be deleted if x was assigned and propagated *really*. This corresponds to anticipating the immediate effects of real assignments in the two subtrees during local expansion (like the subtrees labelled [0] and [1] in figure 4.9, for example).

Figure 5.1 shows part of an NNF-tree. Node 1 is the LCA of variable x. Dashed lines represent paths of arbitrary length, solid lines a path with length of one, and black dots mark collected LCA-children. Node R is a child of the LCA which does not contain an occurrence of x.

Computing decrease scores is implemented as follows: first x is set to false virtually. All AND-nodes (OR-nodes) which are parents of positive (negative) occurrences are marked as deleted by setting a deletion mark in the node. Further, all negative (positive) occurrences which have AND-nodes (OR-nodes) as parents are marked as deleted. In figure 5.1, nodes 3, 4 and 6 and the occurrence at node 5 would be marked. Each marked node denotes the root of a deleted subtree.

Next, all marked nodes are collected under the requirement that a node which has a marked (that is, virtually deleted) predecessor is discarded and the predecessor is collected instead. The reason is that such marked nodes occur in the subtree of another marked (and virtually deleted) node, thus the marked node is subsumed under the subtree of its marked predecessor. In figure 5.1, node 6, which is virtually deleted, occurs in the subtree of node 3, which is a virtually deleted predecessor, and hence would not be collected.

For each positive and negative occurrence, the path from the occurrence up to the LCA x is traversed by following parent pointers and the highest marked node on the path, that is the one at the lowest level, is collected. Upward traversing is interleaved with clearing marks of virtually deleted nodes along the path. This saves clearing marks in an additional traversal of the expansion-relevant subtree.

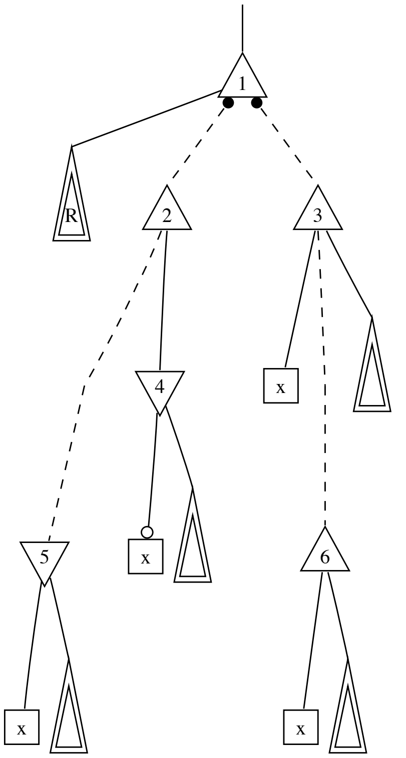


Figure 5.1: Computation of decrease scores – example

It is necessary to check if a marked node to be collected occurs in the collection already. This can happen because such nodes are possibly predecessors of various deleted nodes. Traversing the path starting from all of these nodes will discover one and the same marked predecessor, which should be added to the collection exactly once. For example in figure 5.1, if marked node 3 was a predecessor of various occurrences of x then it would be found more than once during upward traversing. In order to check if a node has been collected already, separate collection marks (not to be confused with deletion marks) are set in all collected nodes. Adding nodes to the collection can then be done in constant time (collections are implemented as stacks).

After node collecting has been carried out for all positive and negative occurrences, the subformula sizes of all collected nodes, that is the numbers of nodes in the respective subtrees, are summed up (the subformula size for each node is maintained and can be looked up in constant time). This yields an intermediate decrease score with respect to assigning the variable false. For the example in figure 5.1, the sizes of nodes 3 and 4 and of the literal at node 5 contribute to the intermediate decrease score.

Setting the variable to true virtually and computing the respective intermediate decrease score is carried out analogously, provided that all deletion and collection marks are reset before.

Finally, the two intermediate decrease scores are added which comes to the decrease score $score_{dec}(x)$ of variable x. Note that it is possible to operate on one and the same subtree (which is the expansion-relevant subtree of x) when virtually assigning the variable true and false because this subtree will be copied if the variable is expanded and real assignments will be made in the respective copies.

Computing decrease scores as described requires $O(nm)$ time, where n is the number of occurrences of the variable and m the maximum node level in the NNF-tree. The path from an occurrence up to the LCA is expected to be short due to the structural restrictions of the NNF-tree which are supposed to keep node levels low.

Alternatively, if these paths were not traversed and occurrences or their parents collected as deleted without searching for virtually deleted predecessors, then more probably than not computing decrease scores would be faster, but at the cost of unreliability of variable scores. Collected nodes which have a virtually deleted and collected predecessor will contribute twice to the decrease score: directly with their own size, and indirectly if the size of the deleted predecessor is added to the incremental decrease score. This could yield optimistic total scores which are no longer an upper bound on the accurate expansion costs because fewer nodes could be deleted in real expansion than predicted.

If the set of collected LCA-children of x contains a literal of x then non-increasing expansion can be applied. In this case, the decrease score has to be computed with respect to the value which really will be assigned to the variable during expansion (see section 4.3.5 on page 59), the opposite value needs to be ignored in virtual assignments. Since the increase score of such variables is zero (see above), the total score will always be negative, thus reflecting a guaranteed decrease of the formula size after expansion.

5.3 Updating Scores

The computation of increase and decrease scores takes into account the number of nodes which are deleted from or added to the NNF-tree during expansions. Therefore, both variable scores and LCAs will have to be updated after any modifications of the NNF-tree, since score computation is based on the LCA of a variable. For example, the expansion-relevant LCA of a variable needs to be updated if the subtree denoted by one of the collected LCA-children does not contain an occurrence of that variable any more.

There are several update strategies. First, scores and LCAs of all variables can be recomputed from scratch. This simple but could become expensive if the number of variables is large.

In a refined approach, recomputation could be done for those variables only, whose scores and LCAs have been affected by the recent modifications of the NNF-tree. This policy reduces the effort of score updates considerably. Particularly on structured formulae, the number of affected variables could be much smaller than the total number of variables in the formula.

Finally, instead of recomputing scores and LCAs from scratch, it is possible to maintain these properties incrementally, which is the most sophisticated but also most complicated approach. For any modification of the NNF-tree, first it has to be determined what property (LCA, increase- or decrease score) of which variable is affected, and second how the new, updated values can be obtained from the old ones. These tasks are not trivial and probably even can not be fulfilled efficiently in the present implementation without introducing auxiliary data structures (see section 5.5 for remarks in this respect).

Based on observations made in this section , a scheme for partial LCA and score update has been implemented in Nenofex which considers only variables whose scores or LCA have been affected by an expansion.

5.4 Marking Variables for Update

The computation of expansion-relevant LCAs and of increase- or decrease scores (directly or indirectly) relies on the literal nodes of a variable. Therefore, adding or deleting literal nodes in an NNF-tree will change the LCA or score of certain variables.

Whenever units or unates are eliminated or a variable is expanded, the NNF-tree is modified. Modifications occur in a subtree, the *modified subtree*, which can be identified depending on the source of modification, for example by figuring out how expansion of a given variable will change the NNF-tree. If a variable is expanded, then the modified subtree consists of the subtree denoted by the expansion-relevant LCA and the parts that have been copied.

In Nenofex, the scores and LCAs of variables are updated based on this observation. By considering the modified parts of an NNF-tree, the set of affected variables is determined. For such variables, any of the properties expansion-relevant LCA, increase- or decrease score could have been changed by modifications, but not necessarily all of them at the same time. For example, there are situations where only the increase score of a variable needs to be updated.

This possibility is taken into consideration by introducing three update marks for affected variables, one for updating the LCA (LU-mark), the increase- and decrease score (ISU-mark, DSU-mark) each. A variable is

- ISU-marked, if it is marked for increase score update
- DSU-marked, if it is marked for decrease score update
- LU-marked, if it is marked for LCA update

Only marked properties are recomputed from scratch. This strategy is a compromise between complex, incremental maintenance of LCAs and scores and simple, but possibly expensive full recomputation of all properties of variables.

5.4.1 Marking for LCA Update

Any time a literal node is deleted or copied during expansion, the respective variable is LU-marked. The first situation can be caught in the function for deleting subtrees, *delete_subformula* introduced in section 3.4.4 (page 30). For the second, LU-marking of variables is interleaved with copying the expansion-relevant subtree if a variable is expanded. This avoids additional traversals of the subtree solely for setting LU-update marks.

The policy for LU-marking is very conservative, particulary with respect to deletions of literal nodes. Deleting an occurrence of a variable does not necessarily invalidate its computed expansion-relevant LCA. This is the case if, and only if, the subtree of at least one collected LCA-child does not contain an occurrence of the variable any more after deletion. For example, in figure 5.1 node 1 is the expansion-relevant LCA of variable x and has two collected LCA-children marked with black dots. Deleting one occurrence of x in the subtree of either LCA-child does not require recomputation of the LCA afterwards (ignoring recursive effects of parent merging and one-level simplification), since both subtrees still contain at least one occurrence.

Every time a variable is LU-marked, it is also ISU- and DSU-marked, because increase- and decrease score change whenever the LCA changes.

5.4.2 Marking for Increase Score Update

Since the increase score of a variable depends on the subformula sizes of its collected LCA-children or of its LCA, a variable is ISU-marked, if one of its occurrences is deleted or copied during expansion because in these situation the expansion-relevant LCA will be recomputed. This policy is similar to LU-marking.

Apart from this, there are situations for certain variables where no occurrences are deleted but which still require the increase scores of these variables to be updated. This must be done every time the subtree of some collected LCA-child of any of these variables is modified, that is its subformula size changes.

In the following two sections, related scenarios are investigated where the increase score of variables is not affected by deletions of one of their occurrences. Finally, based on observations made, the policy of ISU-marking of variables in Nenofex is presented.

In figures 5.2 to 5.6, dashed lines denote paths of arbitrary length and nodes under the requirement of alternating types. Black dots mark LCA-children and nodes R at an LCA of a variable represent subtrees which do not contain occurrences of that variable.

Modified subtrees are defined as the parts of an NNF-tree where modifications will occur if a variable is eliminated.

Inspecting Modified Subtrees

Given a variable x and its expansion-relevant subtree, assume that x is either unit or unate or eliminated by non-increasing expansion, that is no subtrees are copied during the elimination of x (this case is ruled out here because variables would be ISU-marked during subtree copying anyway).

Variable x will be assigned a value and all modifications of the NNF-tree during propagation of the variable will occur in the expansion-relevant subtree of x, which is the modified subtree with root $lca(x)$ this case.

In certain cases, the increase score of some variable y which has occurrences in the modified subtree, assuming that no occurrence of y is deleted during the elimination of x, needs to be updated. Depending on the positions of $lca(x)$ and $lca(y)$, three situations may occur (assume that y has an occurrence in the modified subtree, which is the expansion-relevant subtree of x).

First (see figure 5.2, left tree), assume that $lca(x)$ is a predecessor of $lca(y)$ and that $lca(y) \neq lca(x)$, thus the expansion-relevant subtree of y is contained in the modified subtree. If the expansion-relevant subtree of y contains an occurrence of x then variable y needs to be ISU-marked because there exists an LCA-child of y whose subtree is modified when x is eliminated. In the other case (see figure 5.2, right tree), if the expansion-relevant subtree of y does not contain an occurrence of x then the increase score of y does not have to be updated because the subtrees of its LCA-children are not affected if x is eliminated.

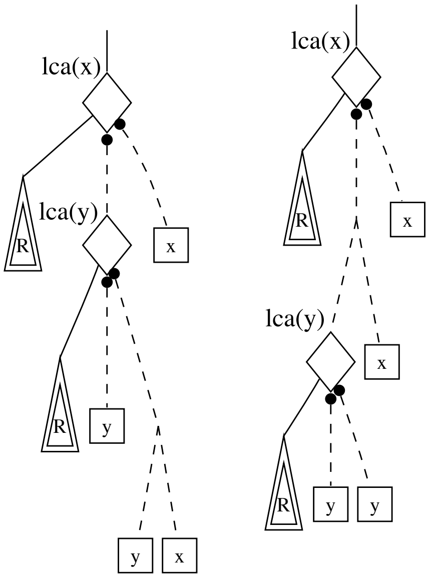


Figure 5.2:

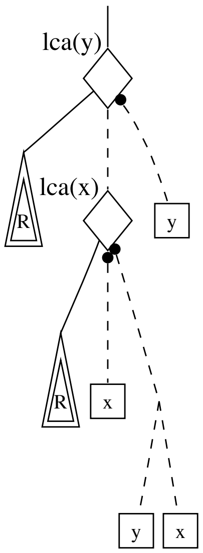


Figure 5.3:

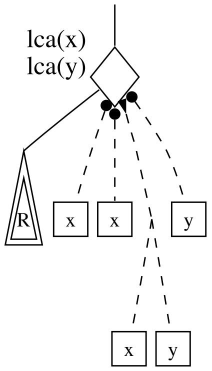


Figure 5.4:

Second (see figure 5.3), assume that $lca(y)$ is a predecessor of $lca(x)$ and that $lca(y) \neq lca(x)$. Since by assumption y has at least one occurrence in the modified subtree, it follows that an LCA-child of y is reachable on a path from that occurrence up to $lca(y)$ by following parent pointers. This path contains $lca(x)$. It is exactly that LCA-child whose subformula size will change because its subtree contains the modified subtree. Hence variable y needs to be ISU-marked.

Third (see figure 5.4), assume that $lca(y) = lca(x)$. Since by assumption y has at least one occurrence in the modified subtree, it follows that the sets of collected LCA-children of x and y are not disjoint (in figure 5.4, the common LCA-child is marked with a black triangle). If x is assigned and propagated then the subtrees of every LCA-child of x will be modified, and so all subtrees of LCA-children common to both x and y. There is at least one common LCA-child, hence variable y needs to be ISU-marked.

In every situation described in this section, variable y had occurrences in the modified subtree. This is not the case for the examples in the following section.

Inspecting Predecessors of Modified Subtrees

Given a variable x and its expansion-relevant subtree, assume that x is either unit or unate or eliminated by expansion, in contrast to the previous section, now

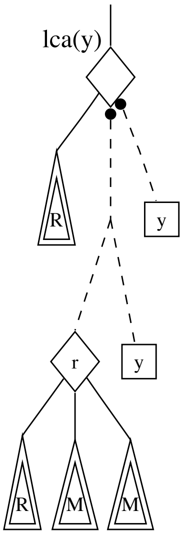


Figure 5.5: ISU-marking of y

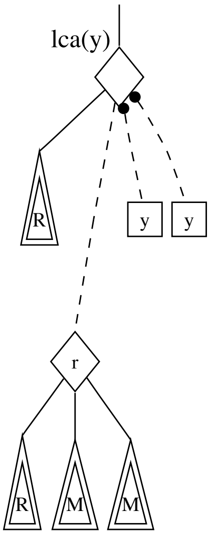


Figure 5.6: No ISU-marking of y

including the possibility of ordinary expansion. In this case, the modified subtree contains the copied subtree as well, otherwise it is the expansion-relevant subtree of x. Let r be the root of the modified subtree (see figures 5.5 and 5.6; nodes M denote parts of the modified subtree).

There are variables which do not have occurrences in the modified subtree but whose increase score still needs to be updated.

For some variable y and its expansion-relevant LCA, assume that $lca(y)$ is a predecessor of r, $lca(y) \neq r$ and y does not have occurrences in the modified subtree (see figure 5.5). If there is a node n on the path $r, \ldots, n, lca(y)$ by following parent pointers such that n is a LCA-child of y, then y needs to be ISU-marked because the modified subtree is a successor of an LCA-child of y and consequently is contained in the subtree of that LCA-child.

Note that it would be too pessimistic to simply check the path for nodes which are the LCA, not an LCA-child, of some variable. In figure 5.6, ISU-marking of variable y is not necessary because the subtrees of its LCA-children do not change. There is no LCA-child of y on the path $r, \ldots, lca(y)$.

ISU-marking Policy

The policy of ISU-marking of variables in Nenofex is drawn from the observations made in the two previous sections. It is conservative in the sense that variables might be ISU-marked unnecessarily (for situations like in the right tree in figure 5.2, for example). The reason is that in the examples recursive effects of parent merging and one-level simplification are not taken into consideration.

Before elimination of units, unates or variables by non-increasing expansion, first the variable's expansion-relevant LCA (which is the modified subtree in this case) is traversed and all variables which have occurrences in this subtree are ISU-marked. Second, the path from the LCA up to the root of the NNF-tree is traversed and all variables which have an LCA-child on the path are ISU-marked. Finally the variable is eliminated.

Concerning ordinary expansion, all variables where occurrences have been copied are ISU-marked (marking is interleaved with subtree copying). Like before, the path from the LCA up to the root is inspected before expansion is carried out.

Additionally, variables where occurrences are deleted are ISU-marked (similar to LU-marking).

5.4.3 Marking for Decrease Score Update

For some variable x, the decrease score depends on the subtree sizes of nodes which have occurrences of x as children because these sizes are summed up during computation (see section 5.2.2). The deletion of any subtree of the NNF-tree will affect the decrease score of variables where a predecessor of the deleted subtree has an occurrence of that variable as a child, since the subtree of the predecessor contains the deleted subtree.

Subtree deletions occur frequently in Nenofex and each time the child lists of predecessors had to be inspected in order to mark all affected variables. In contrast, a conservative approach has been implemented similar to ISU-marking.

Before elimination of units, unates or variables by non-increasing expansion, the variable's expansion-relevant subtree (which is the modified subtree in this case) is traversed and all variables which have occurrences in this subtree are DSU-marked. The path from the modified subtree's root up to the root of the NNF-tree is inspected. For any node on this path which has a literal of some variable x as a child, x needs to be DSU-marked.

In figure 5.7, node 3 is the LCA of some variable v to be eliminated (its occurrences are not shown) and hence is the root of the modified subtree. Node R does not contain occurrences of v, nodes D indicate subtrees to be deleted, dashed lines represent paths of arbitrary length and nodes and solid lines for paths of length one. On the path $3, 2, 1$ node 2 has literals of variables x and y as children, con-

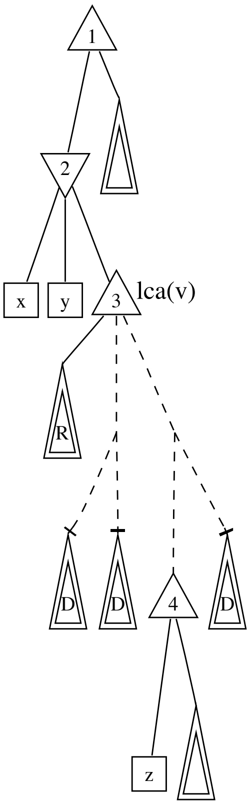
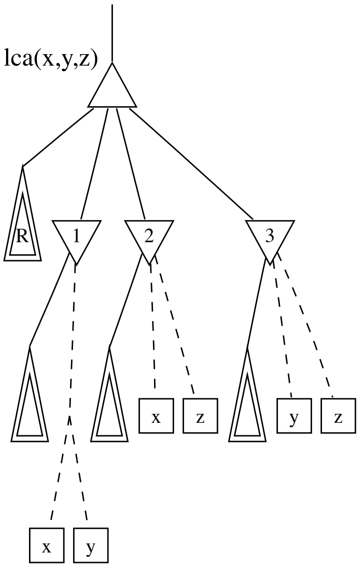


Figure 5.7: DSU-marking of x and y



LCA-child list occurrences	
node 1:	x, y
node 2:	x, z
node 3:	y, z

Figure 5.8: Example for LCA-child list occurrences

sequently these variables are DSU-marked. Note that if decrease scores of x and y are computed, then node 2 will be marked as deleted if variables are virtually assigned true. The subtree of node 2 contains the modified subtree. Therefore the contribution of node 2 to the intermediate decrease score of variables x and y changes.

Concerning ordinary expansion, further all variables where occurrences are deleted or copied are DSU-marked and path checking is carried out before. A path check is necessary because, as described for node deletions, copying subtrees during expansion changes the size of any predecessor's subtree. For predecessors which have literals as children, the respective variables need to be DSU-marked.

All deletions during expansions occur in the modified subtree, which is the least common ancestor of all deleted subtrees (this is node 3 in figure 5.7). The path from its root up to the root of the NNF-tree (nodes $3, 2, 1$) would be traversed multiple times if path inspection was carried out starting from each deleted subtree. This is avoided in Nenofex by inspecting this path once and instead DSU-marking all variables in the modifed subtree as a precaution, where variables might be DSU-marked unnecessarily. In figure 5.7, variable z does not have to be DSU-marked. If the decrease score of z is computed, node 4 will be marked as deleted when z is virtually assigned false. The subtree of node 4 does not contain a deleted subtree and hence its contribution to the intermediate decrease score does not change.

5.4.4 Efficiency Concerns: New Data Structures

In order to mark variables for increase or decrease score update, the path from the root of the modified subtree up to the root of the NNF-tree must be inspected. Due to the structural restrictions, this path is expected to be short and since literal nodes are stored first in child lists, DSU-marking can be carried out efficiently.

Concerning ISU-marking, checking if a node on the path is an LCA-child of some variable is problematic. The sets of LCA-children of all variables in the formula have to be searched for each node on the path. In the worst case, ISU-marking requires $O(mnc)$ time, where m is the maximum node level in the NNF-tree, n is the total number of variables in the formula and c is the maximum cardinality of all sets of LCA-children of variables. In particular large values of n could turn path inspection into a serious performance bottleneck. In order to cope with this problem, two new data structures are introduced.

First, variables which have one and the same node as LCA are stored in a doubly linked list. These lists are implemented the same way as child list: pointers to the previous and next list entry are embedded in each variable object and each node in the NNF-tree has a pointer to the first and last entry of its variable list. At most n nodes can have such a variable list (probably fewer in practice), pointers of remaining nodes are set to null. Embedding pointers in variable objects is possible because a variable occurs in exactly one list if its LCA has been computed.

During path inspection, these variable lists can be used to restrict the number of variables whose LCA-children are searched: if the parent of a path node is the LCA of one or more variables, then it suffices to search the LCA-children of the variables stored in its variable list, rather than of all variables in the formula. However, this is no remedy for the need to search LCA-children.

Searching can be avoided at all if, for each node in the NNF-tree, the set of variables is stored where that node occurs in the set of LCA-children. A node has an *LCA-child list occurrence* in some variable x if it occurs in the set of collected LCA-children of x. Each node has a set of LCA-child list occurrences which are pointers to the respective variable objects. In practice, these sets are relevant only for nodes whose parents are the LCA of at least one variable, and are empty in all other nodes. Note that, in contrast to variable lists introduced above, it is not possible to implement sets of LCA-child list occurrences as doubly linked lists with pointers embedded in variable objects, because a node can be an LCA-child for several variables.

In figure 5.8, nodes 1, 2 and 3 occur in the set of LCA-children of more than one variable. The AND-node at the root is the LCA of variable x, y and z. The table shows LCA-child list occurrences. For example, in node 2 pointers to variables x and z are stored because node 2 is an LCA-child of these variables.

Thus pointers to variables will be added to the set of LCA-child list occurrences of several nodes. Sets of LCA-child list occurrences are implemented as stacks which store pointers to variable objects.

ISU-marking of variables during path inspection can then be carried out by ISU-marking exactly the variables which are stored in a path node's set of LCA-child list occurrences. Searching LCA-children of variables is not necessary any more.

Maintaining the new data structures is a complex issue. Various situations concerning parent merging, node deletions and LCA computation need to be considered. Some questions related to the current implementation are left open. For example, it is not clear if the lists of variables with common LCA introduced first are still needed if sets of LCA-child list occurrences are maintained. Before the LCA of a variable is recomputed, the variable's entries in the sets of LCA-child list occurrences must be removed, which is done by searching these sets in each node which is an LCA-child of that variable. This could possibly become expensive if a variable has many LCA-children whose sets of LCA-child list occurrences are large.

5.5 Future Work

Instead of marking variables for recomputing LCAs, increase- or decrease scores from scratch, these properties could be maintained incrementally if the NNF-tree is modified. The approaches presented in the following sections should be regarded as first, rough ideas for incremental maintenance.

5.5.1 Maintaining LCAs

Concerning node deletions, the LCA and LCA-children of some variable x do not change as long as the subtree of each LCA-child contains at least one occurrence of x (the crucial property of LCA-children). This can be determined efficiently if an occurrence counter is maintained for each collected LCA-child. Each occurrence has a pointer to the LCA-child whose subtree contains that occurrence (alternatively, this LCA-child could be found out via following parent pointers until the

LCA of the variable is reached). If an occurrence is deleted, the counter of its LCA-child is decreased by one. If it goes down to zero, the LCA-child can be remove from the set of LCA-children. The LCA of a variable needs to be recomputed from scratch if, and only if, there is exactly one LCA-child left.

If a variable is expanded, occurrence counters and sets of LCA-children can be maintained depending on the expansion case.

5.5.2 Maintaining Scores

Path inspection could be applied for incremental maintenance of scores. If a subtree of size s is deleted then all predecessors p of its root must visited. If p has a literal of some variable x as a child, then s must be subtracted from the decrease score of x. If p is an LCA-child of some variable x, then s must be subtracted from the increase score of x.

If some variable x is expanded, the sizes of the copied subtrees have to be added to the increase scores of all variables where an LCA-child is a predecessor of the expansion-relevant LCA of x. If a predecessor has a literal of some variable y as a child then the sizes have to be added to the decrease score of y.

The sizes of copied nodes which have a literal node of some variable y as a child must be added to the decrease score of y.

Chapter 6

Redundancy Removal

Local expansion has been introduced in chapter 4 as a method for expanding a variable where only the relevant parts of a formula are copied. This way, introducing redundancy related to unnecessarily copied parts is avoided, but nonetheless the expanded formula likely will contain redundant parts. The same problem applies to expansion-based QBF solvers which operate on CNF. In Quantor [Bie04], redundant clauses are detected using subsumption checking.

In this chapter, the implementation of two approaches related to redundancy removal on NNF will be presented. The first approach has been taken from the domain of *automatic test pattern generation (ATPG)* for combinatorial circuits and is concerned with redundancy removal in general. The second approach has been adopted from circuit optimization: in *global flow* a circuit is minimized by applying transformations which are based on implications. Implications are derived by analizing the global flow of values in the circuit.

Redundancy removal in Nenofex is an orthogonal method for heuristically improving the performance of the solver, but is not an integral part of it. The general approaches of ATPG-based redundancy removal and global flow have been modified and are closely related to each other. Applying only global flow as implemented in Nenofex does not necessarily reduce the size of the NNF-tree, but could enable ATPG-based redundancy removal to detect further redundant parts.

6.1 Preliminaries

In the following two sections, an brief overview on ATPG-based redundancy removal and global flow is provided. The purpose is to introduce the basic concepts as a background for the approaches that have been implemented in Nenofex. The material has been selected from [MLB00] and [KS97].

6.1.1 ATPG-based Redundancy Removal

In a combinatorial circuit various kinds of defects might occur, particularly during the manufacturing process or during the use of the device. A defect causes the circuit to produce wrong output values. The presence of erroneous outputs, that is erroneous behaviour, can be detected by testing. A test is a set of input values, called *test pattern*, where the good and the faulty circuit show different output values.

In *structural testing*, the internal circuit structure (gates, connections between gates) is taken into consideration. In order to test if a particular gate or line is faulty, a *minimal* set of input values is generated by which erroneous circuit behaviour related to that particular part can be detected. Test patterns can be generated algorithmically, which is the main purpose of methods for ATPG.

In testing, certain scenarios of faults are modelled. A typical model for faults related to single connections between gates (not for gates themselves) is the *stuck-at-fault model*. Such faults are modelled by assigning a fixed value to a line. A line is said to be stuck-at-1 (stuck-at-0), if it always carries true (false) regardless of the intended value. A test for a stuck-at fault is a minimal set of input values where the good and the faulty circuit show different behaviour. If a fault does not change the behaviour of the circuit, then it is redundant and the respective hardware may be removed from the circuit without affecting its function.

Detection and removal of redundant stuck-at faults can be combined with ATPG in order to optimize a circuit for size. Faults are tested in cyclic fashion where redundant faults are removed until saturation. Removing redundant faults can cause further faults to become redundant.

In ATPG-based redundancy removal, testing a stuck-at fault comprises three steps:

- *fault sensitization:* the corresponding line is assigned the opposite value of the fault: for a stuck-at-1 fault, the line is assigned false, otherwise true. This is necessary in order to activate the fault. For example, a stuck-at-0 fault can not be detected by a test pattern if the line carries value false anyway.

- *path sensitization:* the effect of the activated fault must be propagated unambiguously along a path, called *fault path*, to an output signal of the circuit. At the output, observed wrong behaviour must be caused solely by the fault. This can be achieved by assigning conservative values to all off-path inputs of gates along the fault path. For example, off-path inputs of OR-gates (AND-gates) must be assigned false (true). This guarantees that the value of gates on the fault path depends on the value at the fault site resulting from fault sensitization. There might be exponentially many paths where

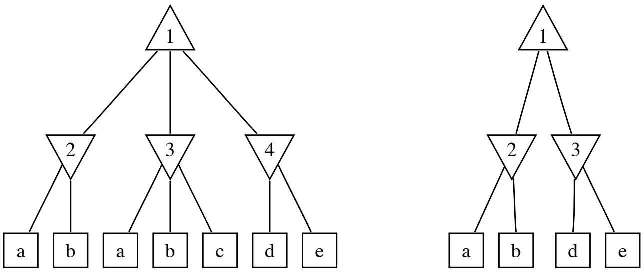


Figure 6.1: Detecting subsumed clauses by ATPG-based redundancy removal

the fault effect can be propagated. If propagation on one path fails, then all remaining paths have to be considered.

- *justification:* all signal assignments made in the two previous steps must be justified by finding a set of input values of the circuit which establishes the configuration of internal signal assignments. This is done by starting at an unjustified, assigned signal and recursively assigning inputs of this signal with justifying values. For example, an AND-gate which is assigned false may be justified by assigning false to one of its inputs. Selecting inputs to be assigned during justification requires making *decisions*. During justifications, conflicts can occur if some signal assignment contradicts a previously made assignment. In this case, conflicting assignments have to be discarded and alternatives have to be chosen. This is referred to as *backtracking*.

If all fault paths and alternative assignments have been tried out but conflicts could not be resolved, then the fault is untestable: it is not possible to find a set of input values such that the fault effect can be observed at an output of the circuit. The corresponding hardware is redundant and may be removed from the circuit without affecting its function.

Details about various ATPG algorithms and a historic survey about their development and refinements may be found in [MLB00].

An example for detecting redundancy is shown in figure 6.1. The tree on the left is a CNF where clause $(a \vee b \vee c)$ is subsumed by clause $(a \vee b)$. The stuck-at-1 fault at the line between node 1 and node 3 is not testable: node 3 is assigned false (fault sensitization) which causes variables a, b and c to be assigned false (justification of false at node 3). Path sensitization requires both node 2 and node 4 to be assigned true, which can not be achieved for node 2 since both a and b are false. Backtracking is not possible, hence node 3 is redundant. The reduced tree is shown on the right.

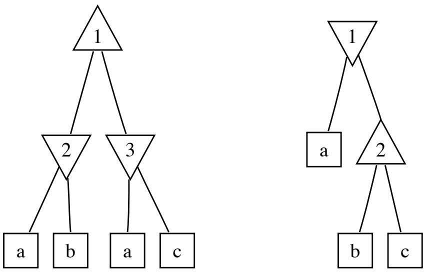


Figure 6.2: Detecting distributivity by global flow

In section 6.2, the implementation of ATPG-based redundancy removal in Nenofex is presented. The special topological situation of an NNF-tree has been taken into account in order to adapt the general ATPG-algorithm.

6.1.2 Global Flow

Global flow is an approach for circuit minimization where implications are derived from signals which are then used to transform and optimize the circuit. Implications provide information about the logical flow of a value. The circuit is transformed in order to reduce its size, but without changing the logical flow of values.

Figure 6.2 shows an example for an optimization which global flow, but not ATPG-based redundancy removal can detect. Literal a may be factored out by applying distributivity. Assigning variable a true will result in assigning node 1 true as well. The tree on the right has been transformed without changing the logical flow between a and node 1.

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. For example, $F_{00}(x)$ is the set of all signals which will be assigned false as a consequence of x being assigned false.

Given the sets F_{VW} for some signal x, the following transformations are valid:

- if $y \in F_{00}(x)$ then replace y by $x \wedge y$
- if $y \in F_{01}(x)$ then replace y by $\neg x \vee y$
- if $y \in F_{10}(x)$ then replace y by $\neg x \wedge y$
- if $y \in F_{11}(x)$ then replace y by $x \vee y$

In order to optimize a circuit, first some signal x is chosen where the sets of implications $F_{VW}(x)$ are computed. In practice, only subsets of the sets $F_{1W}(x)$ and $F_{0W}(x)$ are considered because computation of the full set of implied signals is complex. Furthermore, in these subsets only those signals are taken into account for transformations which are in the *transitive fanout* of x (the fanout of signal x is the set of output signals). In general, circuits are DAGs and some signal x might have several signals in its fanout.

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. After the transformation, it has to be checked if the area has decreased. If not, all modifications are reversed and another node is chosen where again implications are derived. These steps are carried out successively for all signals in the circuit.

Details about the entire process of global flow are omitted here and may be found in [KS97]. In section 6.3, a modified, limited approach which has been implemented in Nenofex is presented.

6.2 Redundancy Removal: Implementation

In Nenofex redundancy is eliminated by testing nodes according to the stuck-at fault model. Regarding the tree as a circuit, testing a node in the NNF-tree means testing the “line” from the node to its parent.

ATPG is an NP-complete problem: testing faults is presumed to require exponential time. For this reason, a limited approach which does not rely on decisions has been implemented in Nenofex. Consequently, backtracking is not needed. The fault type to be tested for a node is chosen such that fault sensitization can be carried out without decisions. Our approach requires polynomial time for testing a node but is incomplete: redundant nodes might remain undetected.

Instead of assigning *nodes* in fault- or path sensitization like in general ATPG, in our approach only *variables* are assigned in these two phases. Assigned variables are then propagated which causes nodes to be assigned. In fault- or path sensitization, variable assignments are drawn from literal nodes which have to be assigned a particular value.

In contrast to testing faults in an arbitrary circuit, where the fault can be propagated to an output on more than one fault path, there is *exactly* one such path for each node in the NNF-tree (this is a structurally inherent property of trees). Selecting of a fault path where path sensitization has to be carried out is trivial.

A lazy approach of path sensitization has been implemented which is interleaved with propagation of assigned variables: instead of assigning off-path inputs of fault path nodes in advance in order to guarantee that the fault effect can be observed at the tree root, conservative values of such off-path input nodes are forced on demand during propagations. Justification of these forced values again does not involve decisions, but only implications.

Conflicts can not be resolved in our approach because no decisions have been made to derive variable assignments and forced node assignments. Hence from a conflict it can immediately be concluded that the fault is untestable.

In the following description of ATPG-based redundancy removal and its implementation in Nenofex, first necessary data structures and related design decisions are introduced. Next, the focus is put on the process of fault testing, in particular on the combination of lazy path sensitization and value propagation and their optimizations.

6.2.1 Data Structures

In Nenofex, global flow and redundancy removal are applied to a subtree of the NNF-tree, the *optimization-subtree*, which is specified by its root r and all relevant children of r. Note that the optimization-subtree denotes a subformula, which allows the relevant children to be a proper subset of all children of r. This notion is similar to the one of LCA-children of expansion-relevant LCAs.

The nodes of the optimization-subtree are kept on a fault queue, which is implemented as a contiguous, circular array. Variable assignments derived during fault testing are enqueued on a propagation queue. Further, all assigned nodes and variables are collected during propagation. For this purpose, marks are used in order to indicate if an object has already been collected or not. In the following, data structures related to nodes and variables are introduced.

Variable-related Data Structures

The set of variables which occur in the optimization-subtree will likely be a proper subset of the set of variables in the NNF-tree, because the optimization-subtree is expected to be smaller. Subtree variables are collected in an initialization phase (see below) and stored separately on a list. Further, each variable has two additional collections of positive and negative occurrences in the optimization-subtree, which are not implemented as ordinary, doubly linked occurrence lists (see section 3.4.1 on page 25), but as stacks.

Node-related Data Structures

In ATPG-based redundancy removal and global flow, variables and nodes are assigned values which are propagated in turn. These tasks require certain pieces of information to be stored for each node in the optimization-subtree.

A node has a value which is either *undefined*, *true* or *false*. Depending on the node type and the value, further assignments of nodes can be triggered. For example, assigning an AND-node true will cause its parent, which is an OR-node, to be assigned true as well. An AND-node will be assigned true if all of its children are assigned true. In order to detect such implied assignments efficently, a watcher scheme has been implemented (see section 6.2.7 below). For each node in the optimization-subtree, the number of unassigned children is maintained together with a pointer to an unassigned child. The use of watchers avoids counting all assigned children of a node each time a child has been assigned. Watchers are an integral part of SAT solvers [ZM02b].

The value of a node is *justified* if, and only if, it is a consequence of the current values of its children. For example, true is justified at an OR-node if at least one of its children is true. Each node has a justification mark indicating if its value is justified or not, and a path mark which is set if the node occurs on the path from the fault node currently being tested up to the root of the optimization-subtree (the fault path).

All mentioned pieces of information are grouped together to form an *ATPG-Info* object. This object is not part of a node in the sense of being embedded in its memory region, but assigned to each node in the optimization-subtree by setting a pointer from the node to the object, which has been allocated in advance.

Embedding ATPG-Info objects into nodes or not is a major design decision. In the first case, memory requirements will be higher because the optimization-subtree is expected to be much smaller than the NNF-tree in practice (in our implementation, an ATPG-Info object has a size of 28 bytes, assuming 4 bytes for pointers and integers). ATPG-Info objects of nodes not in the subtree remain unused during optimization. On the other hand, cache performance during value propagations will likely be better if these objects are embedded. Experiments have not yet been carried out with this respect, but it has been decided to allocate and assign ATPG-Info objects on demand during initialization.

Redundancy removal and global flow operate on a fault queue, which, for example, is a collection of pointers to all nodes in the optimization-subtree. If a node in the subtree is deleted and memory allocated to the node released, it will have to be removed from the fault queue in order to prevent invalid memory accesses in forthcoming test cycles. It would be cumbersome to remove deleted nodes from the fault queue right before they are physically released, because the fault queue had to be linearly searched for the deleted node and for all of its successors, which consequently will be deleted as well.

In order to overcome this situation, a scheme for lazy removal of deleted nodes has been implemented. Instead of storing pointers to real subtree nodes, the

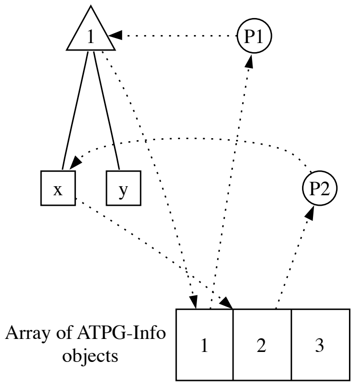


Figure 6.3: Relation between tree nodes and their proxies

fault queue stores pointers to *proxy nodes*. The same applies to the collections of a variable's subformula occurrences. During initialization, each node in the optimization-subtree is assigned an ATPG-Info object and a proxy node: the proxy has a pointer to the real node, the real node a pointer to the ATPG-Info object, which has a pointer to the proxy (this is necessary in order to deallocate proxies of deleted real nodes after redundancy removal has finished; see next section). Memory regions allocated for proxy nodes are different from those of real nodes. Thus deleting and releasing a real node does not invalidate its proxy, which afterwards can still be accessed without causing invalid memory reads. A proxy node has a pointer to the corresponding real node and a deletion mark which is set if the real node has been deleted in either function *merge-parent* or *delete-subformula* (see sections 3.4.4 and 3.4.5 on page 30). During fault testing, proxy nodes to be tested are taken from the fault queue. Before a real node is accessed from its proxy, the deletion mark is checked. If it is set, then the real node has been deleted, hence its proxy can be discarded and another proxy is taken from the queue instead. This way, the fault queue is cleaned up on-the-fly without the need for searching deleted nodes.

Figure 6.3 shows the pointer structure (dotted lines) between tree nodes, its proxies and ATPG-Info objects. A node has a pointer to its ATPG-Info object, which has been allocated in advance in an array (shown at the bottom). Proxies (nodes $P1$ and $P2$) are accessible from an ATPG-Info object. The fault queue would store nodes $P1$ and $P2$ instead of 1 and x.

Initialization and Finalization

Before redundancy removal or global flow can be carried out, data structures necessary for fault testing and value propagation must be set up. First, an array

Algorithm 5: collect_fault_nodes

```
Input: optimization-subtree
Result: fault_queue with proxies of nodes
Data: stack fault_stack, queues fault_queue, queue, node cur
enqueue(queue, root)
while queue not empty do
    cur ← dequeue(queue)
    if cur is operator node then
        push(fault_stack, cur)
        foreach child ch of cur do
            enqueue(queue, ch)
    else
        enqueue(fault_queue, cur)
while fault_stack not empty do
    cur ← pop(fault_stack)
    enqueue(fault_queue, cur)
```

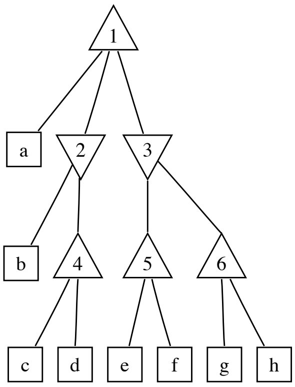
is allocated which stores as many ATPG-Info objects as there are nodes in the optimization-subtree.

Next, the optimization-subtree is traversed. Each node is assigned an ATPG-Info object and a proxy by establishing pointer relations as in figure 6.3. Whenever a literal node is found, the respective variable is added to the list of subtree variables, if not already present (this can be checked in constant time if collected variables are marked). The proxy of the literal node is added to the respective collection of subtree occurrences. Watchers need to be initialized in operator nodes, which is described in section 6.2.7.

Resetting data structures and deallocating proxy nodes does not require the optimization-subtree to be traversed again. Instead, the array of ATPG-Info objects is traversed and for each object, first the real node is accessed via its proxy, provided that the deletion mark is not set, and the pointer to the corresponding ATPG-Info object is set to null (see figure 6.3). Next, the proxy is deallocated. After these tasks have been carried out for each ATPG-Info object, the array can be released. Additionally, sets of subformula occurrences for each collected subformula variable are reset, and finally the set of collected variables as well.

6.2.2 Collecting Fault Nodes

After data structures have been set up, the proxies of all nodes in the optimization-subtree are collected in the fault queue. Fault nodes are tested in the order they appear on the fault queue. Generally, the order does not matter but for



before line 10 in algorithm 5:
fault stack: 1,2,3,4,5,6
fault queue: a,b,c,d,e,f,g,h

finally:
fault stack: ∅
fault queue: a,b,c,d,e,f,g,h,6,5,4,3,2,1

Figure 6.4: Collecting fault nodes

our approach of ATPG-based redundancy removal and for global flow, choosing a particular order could sometimes postpone unnecessary tests if promising nodes are tested first. Literal nodes should be tested before operator nodes because at least the assignment of the literal's variable can be derived from fault sensitization (possibly more during lazy path sensitization; see below). For operator nodes which do not have literal nodes as children, no variable assignments can be derived during fault sensitization, hence such nodes should be tested last.

Choosing the order of fault nodes is a heuristic approach and does not guarantee performance improvements. A simple algorithm for establishing an order has been implemented.

Proxies of nodes are collected in the fault queue by traversing the optimization-subtree in breadth-first search and enqueuing literals before operators, which in turn are enqueued in reverse order as being visited. In algorithm 5, breadth-first search starts at the root of the optimization-subtree (line 1). If an operator is found (line 4), it is pushed on an intermediate fault node stack (line 5), and all of its children are enqueued. Literals are immediately enqueued on the fault queue (line 9). After all nodes have been visited, the fault queue contains literal nodes only. Operators from the intermediate stack are enqueued (line 10), which establishes the final order of nodes on the fault queue.

Figure 6.4 shows an example, where the tree is the optimization-subtree. In line 7 of algorithm 5, children are always enqeued starting at the leftmost child. On the right, first the contents of the intermediate fault stack and of the fault queue are shown after the tree has been traversed, and then the final order of nodes on the fault queue.

```
Algorithm 6: test_fault_nodes
   Input: fault_queue with proxies of nodes
   Result: redundant (untestable) fault nodes
   Data: queues fault_queue, non_red_faults, node fault_node,
         flag test_again, global flag conflict
 1 conflict ← 0, test_again ← 1
 2 while test_again do
 3     test_again ← 0
 4     while fault_queue not empty do
 5         fault_node ← dequeue(fault_queue)
 6         if fault_node marked as deleted then
 7             continue
 8         if is_redundant(fault_node) then
 9             assert(conflict = 1)
10             test_again ← 1, conflict ← 0
11             collect_fault_path_nodes(fault_node)
12             delete_subformula(fault_node)
13         else
14             assert(conflict = 0)
15             enqueue(non_red_faults, fault_node)
16         unmark_fault_path_nodes()
17         reset_touched_nodes_and_vars()
18     if test_again then
19         swap(fault_queue, non_red_faults)
```

6.2.3 Testing Fault Nodes

The nodes on the fault queue are tested successively in cyclic fashion, from the first to the last node, because eliminating redundancies can cause further nodes to become redundant. Whenever redundancy is detected and eliminated, nodes which have not been tested in the current test cycle are processed before the next cycle starts. If testing always continued at the first node on the queue each time a redundant node is found, then some nodes would be tested more than once in a cycle, which should be avoided.

Algorithm 6 shows the steps involved in testing. An iteration of the outer loop (line 2) corresponds to a test cycle where all nodes on the fault queue are tested exactly once. Nodes are tested in the inner loop (line 4). A proxy is taken from the fault queue. If its deletion mark indicates that the corresponding real node has already been deleted, then the proxy is discarded immediately and a new one is dequeued (lines 6 and 7; the loop condition will be re-evaluated if command `continue` is executed). Redundant nodes detected in function *is_redundant* (line 8) are indicated by setting the global flag *conflict* to true, which arises if both values

```
Algorithm 7: is_redundant
   Input: node fault_node
   Result: global flag conflict is true if, and only if, fault_node is redundant (untestable)
   Data: global flag conflict, node fault_node, type fault_type, queue propagation_queue
 1 assert(conflict = 0), assert(propagation_queue is empty)
 2 if parent(fault_node) is OR then
 3     fault_type ← s_a_0
 4 else
 5     fault_type ← s_a_1
 6 fault_sensitization(fault_node, fault_type)
 7 assert(conflict = 0)
 8 lazy_path_sensitization(fault_node, fault_type)
 9 mark_fault_path_nodes()
10 while conflict = 0 and propagation_queue not empty do
11     var ← dequeue(propagation_queue)
12     propagate_assigned_var(var)
13 return conflict
```

true and false are derived for a variable during propagation (see below). Redundant nodes are deleted from the optimization-subtree (line 12), where deletion marks of proxy nodes are set in function *delete_subformula* and proxies of nodes on the fault path are collected (line 11; the purpose of collecting path nodes is pointed out in section 6.2.8 below). A new test cycle will start after all nodes on the fault queue have been processed because flag *test_again* is set to true. Otherwise, nodes not deemed to be redundant (our approach is incomplete; nodes which are in fact redundant might remain undetected) are put on an auxiliary queue (line 15). After a node has been tested, the values of all assigned nodes and variables are reset and the nodes on the path from the fault node to the root of the optimization-subtree are unmarked (lines 16 and 17). These nodes have been marked in function *is_redundant* (line 8) before (the purpose of marking is pointed out below). After all nodes on the fault queue have been tested (that is the fault queue is empty and the current test cycle has finished) and redundancy has been found (line 18), the auxiliary queue, now containing all nodes not found redundant, becomes the fault queue in the forthcoming test cycle and vice versa. In the following sections, the actual process of testing nodes in function *is_redundant* is described.

6.2.4 Fault Sensitization

In order to test a fault node (algorithm 7), first the fault type is determined which depends on the parent's type (line 2). If the parent is an OR-node (AND-node), the fault node will be tested for stuck-at-0 (stuck-at-1). In our approach of ATPG-based redundancy removal, fault sensitization is the process of deriving variable assignments (not node assignments) from fault nodes and is carried out in function *fault_sensitization* (line 6). For stuck-at-0 (stuck-at-1) faults, variable assignments will be derived from the fault node (if possible) which, when propagated, result in assigning the fault node true (false). Hence if a fault node is tested for stuck-at-0 and is a literal node, then the literal's variable is assigned false (true) if the literal node is negated (not negated). Otherwise, if it is an operator (it can only be an AND-node in this case), then for all of its literal children (if any), the variables are assigned false (true) if the respective literal child is negated (not negated). Assignments for stuck-at-1 faults are derived analogously.

Since no decisions are made to assign operator nodes during testing, it is necessary to choose fault types as described because this allows assigning nodes without making decisions. The motivation for choosing the fault type for literal nodes depening on the type of the parent is different. For example, if a literal node with an OR-node as parent was tested for stuck-at-1 instead of stuck-at-0, then this fault would be equivalent to the stuck-at-1 fault of the parent (as pointed out, this fault type has to be tested there), that is the two faults have the same effect: if the literal node is always true (stuck at true), then so is the parent because it is an OR-node. The situation where the literal is stuck-at-1 is already covered by the parent being tested for stuck-at-1. Hence the literal node is tested for stuck-at-0. For literal nodes with AND-nodes as parents, the analogous argument applies.

Variables assigned in fault sensitization are enqueued on the propagation queue. Literal children of a node can be accessed efficiently because they occur before operator nodes in a child list.

Fault sensitization is the motivation for ordering nodes on the fault queue as described in section 6.2.2. Literal nodes are tested first because during fault sensitization the assignment of the literal's variable can always be derived, and possibly more during path sensitization. The more assignments are derived, the more likely a conflict occurs during propagation.

Note that fault sensitization as described can never yield conflicts (line 7) because all nodes in the tree are one-level simplified, in particluar child lists of nodes do not contain complementary literals.

6.2.5 Lazy Path Sensitization

Path sensitization in general derives mandatory node assignments from *all* nodes on the fault path currently being considered. The classical approach could become problematic if child lists are long, like at the root of a large formula in CNF: the full child list had to be traversed and each child had to be assigned a conservative value. This must to be carried out each time a fault is tested.

In order to cope with this problem, a lazy version of path sensitization (line 8 in algorithm 7) has been implemented in Nenofex where variable assignments are derived in contrast to node assignments as in classical path sensitization. Instead of assigning all children of fault path nodes in advance, the path is traversed and, for each path node, only literal children are considered (like in fault sensitization, literal children can be accessed efficiently). If the path node is an OR (AND), then the literal child's variable is assigned true (false) if the literal child is negated, and false (true) otherwise. Variables assigned in path sensitization are enqueued on the propagation queue.

Note that, unlike in fault sensitization, conflicting variable assignments might arise during lazy path sensitization, either in combination with assignments from fault-, or solely within those from path sensitization. In such case, the global flag *conflict* will be set to true (line 8).

Nodes on the fault path are marked (line 9), which is an integral part of lazy path sensitization. Variable assignments derived in the previous step guarantee conservative values for literal children of path nodes only. Values for remaining operator children are taken into consideration on demand during the process of propagation, which avoids setting these values explicitly by fully traversing child lists of path nodes.

6.2.6 Propagation

If conflicting variable assignments have been derived in fault- or lazy path sensitization, function *is_redundant* returns true (line 13 in algorithm 7) without propagating enqeued variables. If the fault node and all fault path nodes have only operator nodes as children, then the propagation queue will be empty before line 10 and the function returns false. Otherwise, assigned variables are taken from the queue and propagated sucessively (line 12).

In function *propagate_assigned_var*, all positive and negative subformula occurrences of the variable to be propagated (these are stored separately and hence can be accessed efficiently) are assigned values. Occurrences which have been deleted are removed from the collections on-the-fly (like with the fault queue, proxies of occurrences are collected). If the variable is assigned true, then positive (negative) occurrences are assigned true (false), otherwise, if the variable is false, then positive (negative) occurrences are assigned false (true). Propagated variables are collected for resetting after testing has finished (see algorithm 6, line 17). Node assignments are carried out in functions *forward_propagate_falsity* (algorithm 8) and *forward_propagate_truth* (algorithm 12 on page 129), which are called on each occurrence depending on the value to be propagated. In either function, conflicts can occur if values *implicitly* imposed by lazy path sensitization can not be justified (see below). In this case, the global flag *conflict* is set to true which causes propagation to abort immediately and function *propagate_assigned_var* returns (line 12 in algorithm 7). During propagation, assigned nodes and variables are collected for resetting values afterwards (see algorithm 6, line 17).

Forward Propagations

In a forward propagation, a value is assigned either to a literal node whose variable is currently propagated (like in function *propagate_assigned_var*), or to an operator node whose value is implied by the values of its children. The latter is a recursive forward propagation. After assigning a node, it must be checked if values constrained by lazy path sensitization are violated (for example, an OR-node on the fault path has a child which has just been assigned true) or require mandatory assignments of certain nodes: if, for example, an OR-node on the fault path has an AND-child which has all of its children assigned to true except one single child remaining unassigned, then this child must be assigned false in order to justify the mandatory value false at the AND-child. Such implied assignments are referred to as backward propagations (see next section).

Algorithm 8 shows the pseudo-code of function *forward_propagate_falsity* (see algorithm 12 on page 129 for the dual function *forward_propagate_truth*), which is called on some node n to be assigned false. Under the preconditions that conflicts did not occur in prior propagations and that n is unassigned, n is assigned false and its justification flag is set to true (line 3). Forward propagations can only happen if the propagated value is justified. This condition is trivially satisfied for assignments of literal nodes (their values are justified by the value of the corresponding variable), but for operator nodes to be assigned in recursive propagations, various cases have to be distinguished in advance.

If parent p of n is and AND (line 20) then a conflict occurs if p is on the fault path but n is not, because n has just been assigned false (line 23): the global flag *conflict* is set to true, which causes propagation in function *propagate_assigned_vars* (algorithm 7 line 12) to stop immediately. A node is on the fault path if it has been marked before (algorithm 7 in line 9). In the general case, p will be assigned false recursively (line 25). If p has already been assigned false before, then its value is now justified by n (line 28). In the latter case, p has been assigned false in an (in a sequence of) implied backward propagation(s) resulting from lazy path sensitization (see algorithm 9 in the next section).

If p is an OR (line 6), then p either is unassigned or assigned true. It can not be false because in this case all of its children had to be false, hence function *forward_propagate_falsity* would have never been called on n.

In the general case (line 16), if p is unassigned and either both p and pp (the parent of p) occur on the fault path or both not (it is impossible that p is on the fault path and pp is not), then p will be assigned false recursively if false is justified

Algorithm 8: forward_propagate_falsity

```
Input: node n to be assigned false
Result: further propagations (backward or forward) or conflict (flag conflict = 1)
Data: nodes n, ch, p and pp, global flag conflict

assert(conflict = 0)
assert(n unassigned)
assign_false(n), n.justified ← 1
p ← parent(n)
pp ← parent(p)
if p is OR then
    assert((p is unassigned) or (p is true))
    if (p is true and not justified) or
      (p unassigned and not on fault path and pp on fault path) then
        if p has exactly one unassigned child then
            ch ← find_unassigned_child(p)
            backward_propagate_truth(ch)
            if p unassigned and conflict = 0 then
                forward_propagate_truth(p)
            p.justified ← 1
    else if p unassigned then
        if all children of p assigned then
            assert(all children assigned false)
            forward_propagate_falsity(p)
else
    assert(p is AND)
    if n not on fault path and p on fault path then
        conflict ← 1
    else if p not assigned then
        forward_propagate_falsity(p)
    else
        assert(p is false)
        p.justified ← 1
```

by the children's values (line 19). Note that *p* can not have a child which is true (line 18), otherwise *p* would have been assigned true before.

If *p* is true and not justified (line 8), or unassigned and not on the fault path whereas its parent *pp* is (line 9), then it has to be checked whether *p* has exactly one unassigned child remaining. The reason is that in the first case *p* has been mandatorily assigned true in an (in a sequence of) implied backward propagation(s) before, and the remaining unassigned child *ch* (line 11) must be assigned true (line 12) for justifying *p* (all other children of *p* are false). In the second case, the value of the remaining unassigned child is implied in the course of lazy path sensitization, which requires *p* (not on the fault path) to be assigned true because *pp* is an AND on the path (as before, all other children of *p* are false). In both cases, truth is propagated backwards by calling function *backward_propagate_truth* on child *ch* (line 12) of *p*. Afterwards, *p* can be assigned true (line 14) unless a conflict occurred in backward propagation(s).

Backward Propagations

In a backward propagation, a value is assigned either to a variable or to an operator node where, for both cases, the value is implied either by certain values constrained in lazy path sensitization or by the value of the respective parent (for example, assigning true to an AND-node will cause all of its children to be assigned true as well). Backward propagations are always carried out on children of nodes, in contrast to forward propagations where parents are assigned. A backward propagation on a literal child represents an assignment of the literal's variable, which will yield a conflict if that variable has already been assigned the opposite value before. Conflicts in backward propagation *always* result from failed justifications of values at children of fault path nodes. For example justification of an AND-node's value (must be false in this case) whose parent is an OR-node on the fault path could fail because backward propagation on the remaining unassigned child yields a conflict.

Algorithm 9 shows the pseudo-code of function *backward_propagate_truth*, called on some node *n* to be assigned true (pseudo-code of function *backward_propagate_falsity* may be found in algorithm 13 on page 130). As in forward propagations, *n* must be unassigned and conflicts must not have occurred before.

If *n* is a literal node (line 3), then the corresponding variable either has already been assigned but not yet fully propagated (line 8), or has not been assigned (line 5). In the first case, a conflict occurs if the variable's value would cause the literal node to be assigned false in forthcoming propagations (line 10). Otherwise (line 5),

Algorithm 9: backward_propagate_truth

```
   Input: node n to be assigned true
   Result: further propagations (backward) or conflict (flag conflict = 1)
   Data: node n, variable var, global flag conflict, queue propagation_queue
1  assert(conflict = 0)
2  assert(n unassigned)
3  if n is a literal then
4      var ← variable(n)
5      if var unassigned then
6          if n negated then assign_false(var) else assign_true(var)
7          enqueue(propagation_queue, var)
8      else
9          if (n negated and var true) or (n not negated and var false) then
10             conflict ← 1
11 else if n is AND then
12     assign_true(n)
13     forall children ch of n and conflict = 0 do
14         assert(ch is literal or OR)
15         if ch unassigned then
16             backward_propagate_truth(ch)
17     n.justified ← 1
18 else
19     assert(n is OR)
20     assign_true(n)
21     if n has exactly one unassigned child then
22         ch ← find_unassigned_child(n)
23         backward_propagate_truth(ch)
24         n.justified ← 1
```

the variable is assigned a value which will (during propagations) result in assigning the literal node true (line 6). The assigned variable is enqueued in the propagation queue.

If n is a an AND-node (line 11), then n and all of its children are assigned true (lines 12 and 16) unless a conflict occurs during recursive backward propagation on any child (line 13) where function *propagate_assigned_vars* would return immediately (algorithm 7 line 12). A child *ch* (line 15) is either already assigned true or unassigned, but not assigned false because in this case n had been assigned false as well and function *backward_propagate_truth* would have never been called on n. After all children have been assigned, the value of n is justified (line 17).

If n is an OR-node (line 18), then it is assigned true in any case (line 20) but justified if, and only if, there is exactly one unassigned child which can be assigned true in turn (line 23) for justification of n (line 24). Such assignments are implied by the values of n and of children already assigned. A child ch (line 22) is either already assigned false or unassigned, but not assigned true because in this case n had been assigned true as well and function *backward_propagate_truth* would have never been called on n.

Remarks

The implementation of forward and backward propagation in Nenofex is non-recursive. Concerning functions *forward_propagate_truth* and *forward_propagate_-falsity*, tail-recursive calls like in lines 19 and 25 of algorithm 8 can easily be avoided by assigning p to n and jumping to the beginning of the function. This simple solution is not possible in an implementation of backward propagation: an auxiliary stack is used where all nodes to be assigned are pushed onto (like in line 16 of algorithm 9).

6.2.7 Watchers

Figuring out the number of assigned children left like in algorithm 8 (lines 10 and 17) or algorithm 9 (line 21) can become a bottleneck if the child list is searched explicitly each time. For example, if the optimization-subtree is a whole CNF, then the child list of the root can be very long.

For this reason, a counter-and-watcher scheme has been implemented which allows both to determine the current number of unassigned children and to retrieve the remaining unassigned child of a node in constant time. Counters and watchers are updated each time a child is assigned, which requires a total of linear time over the *full* sequence of child assignments of a node. Watchers play an important role in efficient SAT solvers. In [ZM02b], various related approaches are surveyed.

Each node has a counter which is initially set to the number of children. Each time a child is assigned, the counter is decremented by one. Thus a node has exactly one unassigned child left if, and only if, the counter has value one. In order to access this child in constant time, a pointer to an unassigned child, the watcher, is maintained in the sequence of child assignments for each node. Initially, the watcher is set to the first child of a node by convention. If the watcher is assigned, then an unassigned sibling must be found by following next-pointers in the child list starting at the current watcher. This is crucial: searching *must not* start at the beginning of the child list each time. The unassigned child becomes the new watcher.

After a node has been tested (algorithm 6 in line 8), counters and watchers are reset to their initial values when assigned nodes and variables are reset (algorithm 6, line 17). This requires nodes where a child has been assigned (and hence the counter or watcher been modified) to be collected in addition to all assigned nodes.

The implementation of the watcher scheme becomes complicated by handling a special situation: since the optimization-subtree denotes a subformula, not all children of its root necessarily occur in this subtree. A typical example is a CNF where only a few clauses are part of the optimization-subtree, whose root is the root of the CNF. The counter-and-watcher scheme must consider only such children of the optimization-subtree's root in order to work properly. Thus at the root (and only there), it must operate locally to the set of relevant children instead of operating on the full child list. This is done by maintaining a *watcher list* where proxy nodes of all relevant children of the root are collected. The list is set up during initialization when ATPG-Info objects are assigned. Deleted nodes must be removed from the list on demand, hence proxies are collected instead of real nodes which allows checking the deletion mark of the proxy. If a node on the watcher list is deleted in parent merging, then new nodes must be added to the watcher list.

The watcher of the optimization-subtree's root is initially set to the first child in the watcher list (instead of the child list). For updating this watcher, an unassigned child on the watcher list must be found which, in contrast, can not be done by following next-pointers as described, but by searching the watcher list starting at the position of the current watcher. Hence the position of the watcher on the watcher list must be maintained together with the pointer to the watcher.

6.2.8 Marking Variables for Update

Deleting redundant nodes in the optimization-subtree modifies the NNF-tree, which requires affected variables to be marked for update. Approaches as described in chapter 5 are applied. All three update marks (see section 5.4 on page 67) are set for a variable where an occurrence has been deleted. Concerning score update, the situation is slightly different. Traversing the optimization-subtree and setting both ISU- and DSU-marks for all occurring variables is likely be too conservative, that is many variables could be marked unnecessarily. Furthermore, the extent of modifications caused by redundancy removal is expected to be smaller than the one caused by expansions.

For this reason, the approaches for marking variables have been adapted. For each redundant node to be deleted, the proxies of nodes on the path from the redundant node up to the root of the optimization-subtree are collected (see algorithm 6, line 11). Instead of traversing the optimization-subtree and consequently considering all nodes, marking is restricted to the set of collected path nodes because these nodes all are predecessors of deleted subtrees. For each collected path node which has a literal node as a child, the corresponding variable is DSU-marked. If a path node has LCA-child list occurrences in some variables, then these are ISU-marked. The same arguments apply as pointed out in sections 5.4.2 and 5.4.3.

Additionally, the nodes on the path from the root of the optimization-subtree up to the root of the NNF-tree are considered and marks are set as described for collected path nodes before. The nodes on this path are ignored during collection because the path had to be traversed for every redundant node (all the path nodes are predecessors of every redundant node).

Marking both with respect to collected nodes and path nodes up to the root of the NNF-tree is carried out once after the whole process of testing nodes (see algorithm 6) has finished.

6.3 Global Flow: Implementation

The implementation of global flow relies on the *same* data structures as redundancy removal (see section 6.2.1): the same tasks for initializing and resetting have to be carried out before and after the whole process of global flow as before and after redundancy removal. Nodes of the tree are *tested* for implications successively, which suggests that our approach is incomplete: not all possible implications can be identified. In order to compute sets of implications for some node in the tree, variable assignments are derived from that node and propagated. Only forward propagations are needed, where the same functions *forward_propagate_truth* and *forward_propagate_falsity* as in redundancy removal are called. Backward propagation can never occur in global flow because there are no mandatory node assignments to be fulfilled.

Our approach is a modified version of general global flow as described in [KS97]. For some node x, the types of implications (see section 6.1.2) considered for transformation are limited to sets $F_{00}(x)$ and $F_{11}(x)$. On the one hand, transformations related to all other types require the introduction of negation, which would violate the structural properties of NNF (for example, transformations related to set $F_{01}(x)$ if x is an OR-node), and on the other hand, not all sets can be computed because in our approach decisions are not made for assigning nodes (for example, computation of sets $F_{10}(x)$ and $F_{11}(x)$ requires decisions if x is an OR-node).

The most important difference between our approach and general global flow is to ignore the option of reversing modifications. Instead, only such modifications are carried out which do not increase the size of the NNF-tree (see section 6.3.2). Furthermore, transformations are applied with nodes already present in the NNF-tree. In a tree there is exactly one path (fault path) from each node up to the

Algorithm 10: find_implications

```
   Input: queue with proxies of nodes
   Result: derived implications
   Data: queues tested_nodes, queue, nodes imp, n, flag test_again
 1 test_again ← 1, imp ← null
 2 while test_again do
 3     test_again ← 0
 4     while queue not empty do
 5         n ← dequeue(queue)
 6         if n marked as deleted then
 7             continue
 8         imp ← derive_implication(n)
 9         if imp not null then
10             collect_implication_path_nodes(n)
11             mark_implicant_vars_for_update(n)
12             test_again ← 1
13             apply_transformation(n, imp)
14             imp ← null
15         enqueue(tested_nodes, n)
16         reset_touched_nodes_and_vars()
17     if test_again then
18         swap(queue, tested_nodes)
```

root. In our approach, the highest implication, that is the one closest to the root, on this path is considered for transformations.

In contrast to the general approach, the application of global flow in Nenofex does not necessarily reduce the size of the NNF-tree. It is the purpose of a subsequent application of redundancy removal to identify parts which have not been detected as redundant before transformations in global flow.

6.3.1 Finding Implications

Algorithm 10 shows the pseudo-code of function *find_implications*. The basic work flow is similar to the one of function *test_all_faults* in algorithm 6: nodes are taken from the queue and tested for implications in cyclic fashion (lines 2, 4 and 17), where deleted nodes are discarded (line 7) as in fault testing. If an implication can be derived from some node n (line 9), then proxies of nodes on the path from n up to the root of the optimization-subtree are collected (line 10) and all variables occurring in the subtree of n are marked for full update of LCA and scores (line 11). The purpose of collecting path nodes and marking is described in section 6.3.3 below. The optimization-subtree is modified (line 13), hence a

Algorithm 11: derive_implication

```
Input: potential implicant n
Result: the highest node implied by n (if any) on the path from n up
        to the root of the optimization-subtree
Data: queue propagation_queue, variable var, nodes n, high_imp
1 assert(propagation_queue is empty)
2 get_necessary_var_assignments(n)
3 high_imp ← null
4 if propagation_queue not empty then
5     while propagation_queue not empty do
6         var ← dequeue(propagation_queue)
7         propagate_assigned_var(var)
8     high_imp ← find_highest_implication_on_path(n)
9 return high_imp
```

new test cycle will start (line 12) if all nodes on the queue have been processed. Transformations related to implications might also cause parent mergings and one-level-simplifications. For this reasons, node n is re-enqueued to be tested again for implications in the next cycle (line 15), no matter if an implication has been derived or not. After a node has been tested, all assigned nodes and variables are reset similarly to fault testing (line 16).

Deriving Implications

Testing nodes for implications is carried out in function *derive_implication*, the pseudo-code of which is shown in algorithm 11. From the potential implicant n (line 2), variable assignments are drawn (if any) which depend on the parent's type *exactly* as if that node had been tested for a stuck-at-fault (see algorithm 7, lines 2 and 6): propagation of assigned variables is then expected to result in assigning n true (false) if its parent is an OR (AND). Note that this need not happen necessarily, for example, if n has only operator nodes as children. In this case, no variable assignments can be derived.

From some node, the *same* set of variable assignments is drawn if that node is tested for a stuck-at-fault as if it is tested for implications (in fact, in the implementation of global flow, function *fault_sensitization*, originally devised for fault testing, is called to derive variable assignments from a potential implicant). The reason is that our adapted approach of global flow entirely relies on propagations, decisions related to fault- or implication testing are not considered at all. For example, variable assignments can not be drawn from an AND-node to be assigned false without making a decision: at least one of its children must be assigned false.

Hence deriving variable assignments from potential implicants in adapted global flow always intends to assign an AND-node (OR-node) true (false) because this can be achieved by assigning all of its children true (false).

The policy of deriving variable assignments based on the stuck-at-fault model is applied to literals as well, but for different reasons. If the potential implicant is a literal node and its parent an AND-node (OR-node), then the corresponding variable will be assigned false (true) if the literal is positive and true (false) otherwise. In the forthcoming propagation of the variable, the parent will then be assigned in a forward propagation.

The assignment of exactly one variable can be derived if the potential implicant is a literal, where generally both true or false could be assigned to the variable independently from the stuck-at-fault model. For example, for a positive literal with an AND-node as parent, the corresponding variable could be assigned true as well in a general approach of global flow instead of false as described before. For literals as potential implicants in our adapted approach, the reason why only variable assignments according to the stuck-at-fault model are considered is related to the transformation of the optimization-subtree (see next section): transformations related to implications are carried out without generating new nodes, that is nodes which are already present in the tree are re-arranged.

Like in fault testing, derived variable assignments are propagated successively (line 7 in algorithm 11) by assigning occurrences depending on their polarities and the value of the variable. On the other hand, backward propagations and conflicts can not occur during propagation in global flow because, unlike in fault testing, there are no mandatory assignments where justification could fail. After all variables have been propagated until saturation in function *propagate_assigned_var*, the path between node n and the root of the optimization-subtree is inspected (line 8), where several cases concerning the values of path nodes have to be handled.

Function *find_highest_implication_on_path* returns null immediately if either n is unassigned or is an OR-node (AND-node) and assigned true (false). In the first case, assigning the potential implicant failed at all where in the two latter cases the wrong value according to the stuck-at-fault model had been assigned (we conjecture that the potential implicant is redundant then and could be removed; in the implementation these two cases are ignored). From all path nodes, the highest node *high_imp*, that is the one at the lowest level, whose value equals the one of the potential implicant is selected. The requirement of equal values ensures that implications of the form $n = 0 \rightarrow \mathit{high\_imp} = 1$ and $n = 1 \rightarrow \mathit{high\_imp} = 0$ are ignored. Transformations related to such implications would add negation operators which had to be eliminated in order to keep the represented formula in NNF. This problem does not arise if only implications $n = 0 \rightarrow \mathit{high\_imp} = 0$ and $n = 1 \rightarrow \mathit{high\_imp} = 1$ are considered.

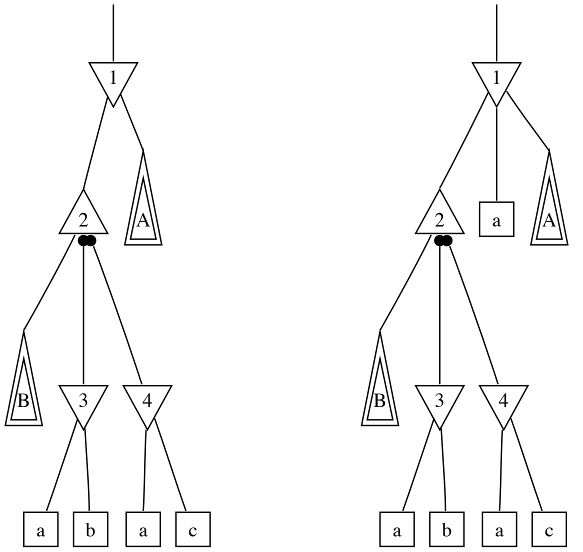


Figure 6.5: An unreliable implication yields an invalid transformation

If an implication has been identified in path inspection, it has to be checked if that implication is justified at a global view. The root of the optimization-subtree might have children which are not part of the subtree (this is also the reason of introducing watcher lists in section 6.2.7). Values at the root could be justified locally with respect to the children in the optimization-subtree, but not when all children are taken into consideration. Figure 6.5 illustrates the problem: node 2 is the root of the optimization-subtree which contains children 3 and 4 (marked with black dots), but not B. If literal a at node 3 is tested for implications, then variable a will be assigned true which results in node 2 being assigned true as well during propagation, because all of its children in the optimization-subtree are assigned true (node 2 has a watcher list). But the value of node 2 is not justified from a global point of view, because child B is not assigned true. Hence the derived implication is *unreliable* and must be discarded since the transformation is invalid (left tree in figure 6.5). Note that unreliable implications can occur at the root of the optimization-subtree only because this is the only node which can have a watcher list. On the other hand, true (false) at the root is always justified if it is an OR-node (AND-node) and derived implications are therefore always reliable.

Node *high_imp* returned by function *find_highest_implication_on_path*, unless it is null, is the highest node implied by n (algorithm 11, line 8) and the corresponding transformation can be applied (algorithm 10 , line 13).

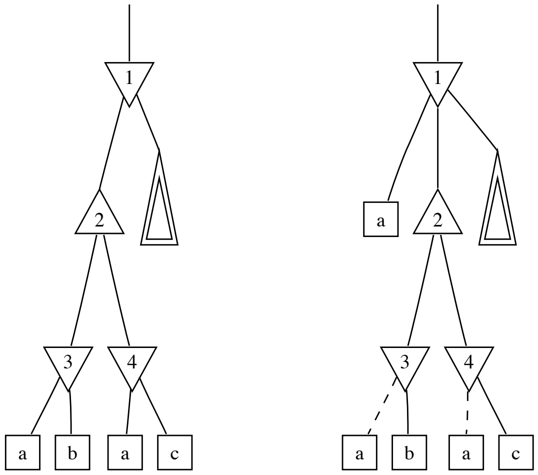


Figure 6.6: Transformation related to literal nodes as implicants

6.3.2 Applying Transformations

In a general approach of global flow, a circuit is modified by inserting and removing particular connections according to derived implications. All modifications are reversed if it turns out that the area of the circuit has been increased. In our adapted approach, the size of the tree is not monitored when transformations are applied and the option of reversing modifications is ignored. Furthermore, only such implications are considered which *do not* increase the size of the tree.

By restricting the set of implications and deriving variable assignments according to the stuck-at-fault model as described, it is guaranteed that transformations need not introduce new nodes but can be carried out by simply re-arranging nodes which are already present in the tree. Transformation for implication $n = 0 \rightarrow high\_imp = 0$ is applied on the tree by replacing $high\_imp$ by $n \wedge high\_imp$, and for implication $n = 1 \rightarrow high\_imp = 1$ by replacing $high\_imp$ by $n \vee high\_imp$ (see section 6.1.2). If new nodes were added to the tree in the transformation in our adapted approach, then the original node n would be redundant in *any* situation, that is the respective stuck-at fault would be untestable. Hence it is valid to re-arrange present nodes instead of introducing new ones and applying redundancy removal to the original node afterwards. In practice, the outcome of testing that stuck-at fault is anticipated, but redundancy removal will have to be applied to the optimization-subtree anyway after global flow has finished because further nodes could become redundant.

Consider the example in figure 6.6 which shows parts of optimization-subtrees. First (left tree in figure 6.6), assume that node 1 is the highest node assigned

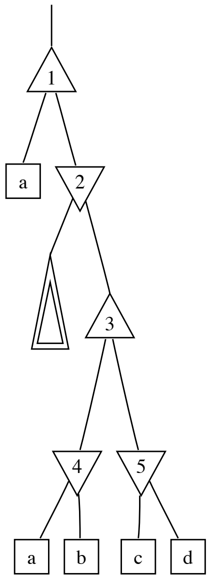


Figure 6.7: Copying vs. moving implicants

true if literal a at node 3 is tested for implications (variable a will be assigned true according to the stuck-at-fault model as described). In the tree on the right, the necessary transformation has been performed: a copy of literal a has been added to node 1. Observe that the stuck-at fault of the original literal at node 3 (also the one at node 4) is now untestable because a conflict will occur in lazy path sensitization at node 1. Dashed lines indicate nodes where the corresponding stuck-at-fault is untestable. Therefore, the original literal at node 3 in the tree on the left could be unlinked and added to node 1 (literal a at node 4 will again be untestable then) without generating a new node.

When assigning variables from literal nodes which are tested for implications, it is necessary to keep the stuck-at-fault model to be able to re-arrange nodes instead of copy nodes. In figure 6.7, if literal a at node 4 is tested for implications and variable a is assigned false (instead of true according to the stuck-at-fault model) then node 1 will be the highest implied node. But *moving* literal a from node 4 to node 1 is an invalid transformation: in this case there is no untestable stuck-at fault the test of which could be anticipated. A *new* literal a had to be added to node 1 in order to transform the tree properly (the stuck-at fault of the original literal a at node 4 is still testable, that is not redundant, then).

For transformations based on implicants which are operator nodes, the same argument applies as for literal nodes. In figure 6.8, node 5, when assigned false, is an implicant for node 1 (variables a and b will be assigned false; assume that node

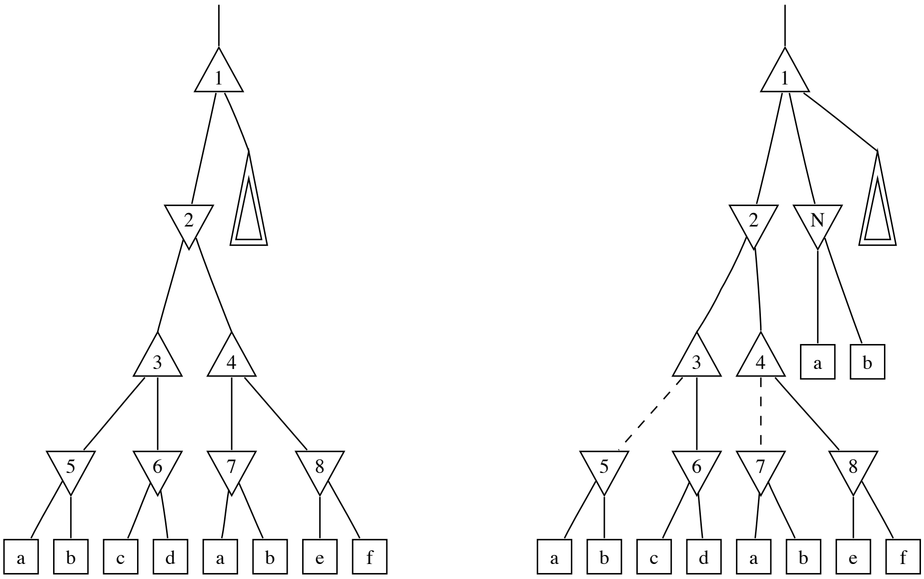


Figure 6.8: Transformation related to operator nodes as implicants

1 is the highest node implied by node 5). Like before, a conflict will occur at new node N in lazy path sensitization if the stuck-at-fault if original node 5 in the tree on the right is tested. Examples for the dual cases may be found in figures A.5 and A.6 on page 128.

Given an implication defined by nodes n and *high_imp*, cases have to be distinguished depending on the type of the implication and on the type of *high_imp* when a transformation to the optimization-subtree is applied. If $n = 0 \rightarrow \mathit{high\_imp} = 0$, then n is a literal or an OR-node and becomes a child of *high_imp* (a sibling *high_imp*) if *high_imp* is an AND-node (OR-node). Otherwise, if $n = 1 \rightarrow \mathit{high\_imp} = 1$, then n is a literal or an AND-node and becomes a child of *high_imp* (a sibling *high_imp*) if *high_imp* is an OR-node (AND-node). Note that the type of n follows from the implication type due to the policy of deriving variable assignments according to the stuck-at-fault model. The structural restriction of alternating types over levels will be kept in any case without taking special steps.

The following actions have to be carried for re-arranging nodes in order to transform the tree:

- unlink n from its old parent *op*
- update subformula sizes of *op* and its predecessors by subtracting *size*(n)
- add n to the child list of its new parent *np* depending on the type of *high_imp* and the implication type

- update the subformula sizes of *np* and of all its predecessors by adding $size(n)$
- update the levels of all nodes in the subtree of n (this subtree is expected to be small)
- if n is a literal then apply one-level simplification to *np*
- if *op* has only one child left, then apply parent merging

If n is added to the child-list of *high_imp* then it must also be added to the watcher list of *high_imp*, if needed.

6.3.3 Marking Variables for Update

Marking variables for LCA- and score update in global flow is very similarly to marking in redundancy removal (see section 6.2.8): again, nodes on the path from n up to *high_imp* are collected for each derived implication (see algorithm 10, line 10) and marks are set as in redundancy removal. Additionally, the subtree of the implicant is traversed and all occurring variables are LU-, ISU- and DSU-marked (algorithm 10, line 11). This is done as a precaution because the LCA of any of these variables could be affected when nodes are re-arranged.

Chapter 7

Putting It All Together

This chapter presents a global view on the system of Nenofex as an expansion-based QBF solver for negation normal form which integrates the approaches introduced in the previous chapters. In section 7.1, the core algorithm is described: after an instance of QBF in QDIMACS format [QDI05] has been parsed and data structures have been initialized, variables are successively expanded in a greedy scheduling policy starting at the innermost scope, where the goal is to keep the NNF-tree small in each expansion. If there are either only existential or only universal variables left, then a propositional formula in CNF is generated from the NNF-tree which is forwarded to a SAT solver. In this case, the result for the QBF instance depends on the result returned by the SAT solver.

Finally, experimental results are presented in section 7.2, where Nenofex is compared against Quantor [Bie04] on benchmark instances from the competitive QBF evaluation 2007 (available from QBFLIB [GNT01b]). Additionally, the effect of global flow and redundancy removal on the performance of Nenofex is investigated.

7.1 System Description

This section presents the components of the core algorithm in Nenofex step by step from initialization of data structures to returning an answer for the given QBF. On certain parts, remarks will be made which are relevant in the context of the whole system, but which have been neglected before in the respective chapters where the parts have been described independently from each other. Figure 7.1 shows the phases of the algorithm. As indicated, in either phase the solver may stop and return an answer for the given QBF.

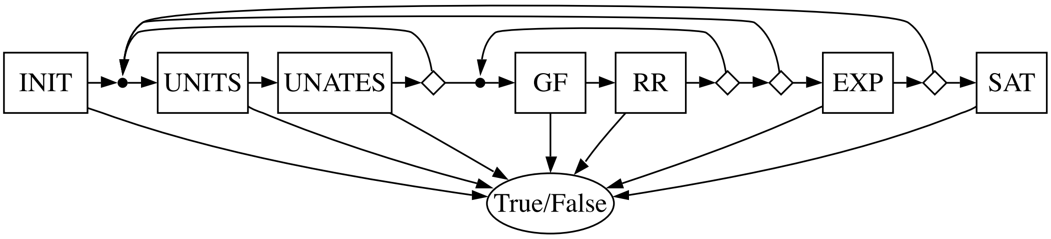


Figure 7.1: Core algorithm of Nenofex. Parsing and initialization (INIT), elimination of units and unates (UNITS, UNATES), global flow (GF), redundancy removal (RR), expansion (EXP) and propositional SAT solving (SAT)

7.1.1 Parsing and Initialization

The first phase (INIT in figure 7.1) involves parsing the given problem instance in QDIMACS format and setting up data structures. Scopes are stored on a *scope list*, where each scope has a *variable list* and a priority queue where pointers to the variables in the scope are stored in ascending order with respect to their score (see section 7.1.4 below). The priority queue is implemented as an array-based heap data structure (for heaps see [OW02], for example).

Clauses are parsed and added to the NNF-tree after one-level simplification has been carried out on each. Node properties like subformula size or level are set on-the-fly. The result of parsing is an NNF-tree which represents the input CNF, which is fully one-level simplified and where all node properties are correctly set. No additional traversals of the tree are needed to carry out these tasks. It may happen that the solver aborts parsing and immediately returns a result. For example, if the input formula contains two complementary unit literals, then it can be concluded that the formula is false during parsing.

Two scope pointers are set, one to the innermost scope, called *current_scope*, and the other one to the first non-empty non-innermost scope, called *next_scope*. Variables will be selected for expansion from either of these two (the actual scheduling policy is described below), but the invariant must be kept up that both scopes are never empty (unless there is exactly one non-empty scope left in the formula) and do not have the same type. If any of the two becomes empty in the course of expanding variables, then scope pointers will be moved in direction towards the outermost scope on demand until the next non-empty scope is encountered. If both *current_scope* and *next_scope* have the same type, then the two scopes are merged into one scope which becomes the new *current_scope* and the pointer to *next_scope* is moved.

Before expansion of variables starts, the scores of all variables in *current_scope* (and only those) are computed and the order in its priority queue is set up.

7.1.2 Elimination of Units and Unates

Unit elimination is the first step in the main loop for eliminating variables in Nenofex, which consists of phases UNITS up to EXP (expansion) in figure 7.1. Units and unates are detected and eliminated until saturation in phases UNITS and UNATES as described in sections 3.4.8 and 3.4.9 on page 36. These two phases are entered in cyclic fashion until the NNF-tree neither contains units nor unates. Jumping back to unit elimination after successful unate elimination in phase UNATES is necessary because unate elimination may produce new units. Like in the previous phase, it may happen that the result for the formula is known during these two phases. For example, if the root of the NNF-tree is an OR-node and has as a child a literal node of a variable which is unate, then the result will be true if that variable is assigned a value such that the literal child becomes true.

7.1.3 Global Flow and Redundancy Removal

Global flow (phase GF in figure 7.1) and redundancy removal (phase RR) are carried out on an *optimization-subtree* which is built depending on recent modifications of the NNF-tree. The optimization-subtree in general denotes the region where all modifications over a sequence of operations (expansions, unit- or unate elimination) have been carried out. For example, if the optimization-subtree is empty and a variable is expanded then this subtree will be exactly the variable's expansion-relevant subtree including copied subtrees and minus deleted nodes. If another variable is expanded, then the optimization-subtree is updated accordingly. The idea is to carry out global flow and redundancy removal on the region of the NNF-tree where modifications have been performed because this region likely contains redundant parts after subtrees have been copied.

The optimization-subtree can become very large over a sequence of expansions. In practice, it is therefore necessary to impose a fixed limit on the size of this subtree and discard parts of it on demand. Consequently, limitting the size will also reduce the runtime spent in phases GF and RR.

Like elimination of units and unates before, global flow and redundancy removal are carried out in cyclic fashion, where each optimization runs until saturation. If redundancy has been found and removed in phase RR, then a new cycle will start at GF because the optimization-subtree has been modified and hence further implications could be derived. This process will continue until neither implications nor redundancies can be identified in the optimization-subtree. If any modifications have been performed in the cycles (consisting of GF and RR) before, then phase UNITS is entered again (see the second branch after phases GF and RR in figure 7.1).

In practice, optimizing large subtrees until saturation with respect to implications and redundancy is impractical due to the amount of required runtime. In order to cope with this problem, fixed, separate limits are imposed on the size of the optimization-subtree and on the number of propagations during global flow and redundancy removal. If a propagation limit is reached, then the respective phase is left immediately and will not be entered again in a cycle (this option is not reflected in figure 7.1) but the other phase remains enabled, provided that its limit has not yet been reached.

The result for the formula is known either in phase GF or RR when the root of the NNF-tree is deleted during one-level simplification.

7.1.4 Expansion

Expansion is the final phase (EXP in figure 7.1) in the variable elimination loop. Variables are selected for expansion depending on their scores and on the types of *current_scope* and *next_scope*. In Nenofex, generally a greedy strategy is applied: in order to keep the size of the NNF-tree small in each expansion, always the *minimum score variable* is selected.

The invariant on the types of *current_scope* and *next_scope* guarantees that these scopes always have different types. Consequently, only two cases can occur.

First, if *next_scope* is existential and *current_scope*, which is the innermost scope, is universal, then only variables from *current_scope* may be selected, that is only universal expansion of variables from the innermost scope is carried out. On formulae in CNF this case can easily be handled by applying forall-reduction to eliminate all variables in *current_scope*. Before selection, the expansion-relevant LCAs and scores of variables in *current_scope* which are marked for update (any of LU-mark, ISU-mark or DSU-mark may be set) are recomputed. Marked variables need not be searched explicitly in this scope, but are collected on a list at the time when they are marked and hence can be accessed efficiently. Generally, only variables from *current_scope* are collected for update. After scores have been recomputed, the order in the priority queue of *current_scope* is updated. The minimum score variable is removed from the priority queue and expanded.

For the second case where *next_scope* is universal and *current_scope* is existential, variables from both scopes may be selected for expansion, hence universal expansion of variables from the first non-innermost scope is the second possibility in this case. But this kind of expansion must first be enabled. A variable from *next_scope* is eligible for expansion if, and only if, the *preceding* expansion (1) caused the size of the NNF-tree to increase and (2) the size increase (not the absolute size) to exceed a threshold, called the *universal threshold*, the value of which is chosen heuristically. The universal threshold is initially set to 10 nodes. If an expansion causes the size of the NNF-tree to increase by 10 or more nodes, then universal expansion of exactly *one* variable from *next_scope* will be *forced* in the next cycle (a cycle consists of phases UNITS to EXP in figure 7.1). After the forced universal expansion, the universal threshold is increased by 10 nodes. Again, the increased threshold must be exceeded to enable another forced universal expansion of a variable from *next_scope*.

The reason for forced universal expansions in the case where *next_scope* is universal and *current_scope* is existential is that score computation of non-innermost universal variables is expensive (depending existential variables in *current_scope* have to be collected). Updating the scores of all affected variables from both *next_scope* and *current_scope* after each expansion can become impractical on large formulae. Consequently, only the scores of variables from *current_scope* are recomputed and when it comes to force a universal expansion of a variable from *next_scope*, the scores of all variables in that scope are computed from scratch and the minimum score variable within that scope is chosen. This typically constitutes a deviation of the greedy strategy of variable selection because the minimum score variable from *current_scope* may well be cheaper than the one in *next_scope*.

In phase EXP, it may happen that the whole NNF-tree is deleted when a variable is expanded. In this case, the solver will immediately return a result.

7.1.5 SAT Solving

After each expansion in the variable elimination loop, it is checked if there are both existential and universal variables left in the formula denoted by the NNF-tree. If not, then the elimination loop terminates and a propositional formula in CNF is generated from the NNF-tree which is in turn forwarded to a SAT solver.

If there are only existential variable left, then a CNF will be generated which is satisfiable if, and only if, the formula denoted by the NNF-tree is *satisfiable*. Thus the SAT solver is used for *satisfiability checking*.

Otherwise, if there are only universal variables left, then a CNF will be generated which is satisfiable if, and only if, the formula denoted by the NNF-tree is *not a tautology*. The SAT solver is used for *tautology checking*. Some formula ϕ is a tautology if, and only if, formula $\neg\phi$, the refutation formula, is unsatisfiable.

The algorithm for generating a CNF from an NNF-tree requires time which is linear in the number of nodes in the tree and is based on the Tseitin transformation [Tse68]. Ideas from the two approaches of Boy de la Tour [dlT92] and Plaisted and Greenbaum [PG86] are combined to reduce the number of clauses in the resulting CNF.

7.2 Experimental Results

Experiments have been carried out on the test set used in the competitive QBF evaluation in 2007 (benchmarks and results are available from [GNT01b]). The test suite contains 1136 instances. 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.

Both Nenofex and Quantor were run on the full set of instances, and for Nenofex, the following three versions concerning global flow (GF) and redundancy removal (RR) have been set up:

- (GF, RR): both global flow and redundancy removal are enabled
- (no GF, RR): only redundancy removal is enabled
- (no GF, no RR): both global flow and redundancy removal are disabled

Additionally, in each test run the size of the optimization-subtree has been limited by 500 nodes in all versions of Nenofex.

Table 7.1 shows a comparison of Quantor and Nenofex in all versions by considering the number of instances where the solvers succeeded, timed out (line OOT) or ran out of memory (line OOM). The number of instances solved by Nenofex decreases from (GF, RR) to (no GF, no RR), which indicates that global flow and redundancy removal contribute positively to the solver's performance. On the other hand, time is traded for memory when enabling optimizations: the number of instances where Nenofex ran out of time increases from (no GF, no RR) to (GF, RR), the opposite effect can be observed concerning the memory limit.

Table 7.3 shows the numbers of instances where either both or only one of Quantor and the best-performing version of Nenofex succeeded, timed out or ran out of memory. The sources of timeouts seem to be different in Quantor and Nenofex, which timed out more often.

Figure 7.6 shows a plot of sorted *penalized* runtimes of Quantor and all three versions of Nenofex. The times recorded for instances where a solver ran out of memory were set to 900 seconds (the time limit) in order to have comparable runtimes (times are plotted on the vertical axis, the unit is seconds).

Figures 7.2 to 7.5 show scatter plots of penalized runtimes in seconds. A data point is defined by the time achieved by two solvers on a particular instance. Vertical and horizontal line-shape cumulations of data points along value 900 indicate instances where one of the two solvers ran out of time or memory.

The conjecture that redundancy removal can profit from global flow is substantiated in several ways. The first line in table 7.2 shows the percentage of nodes which have been deleted in the variable elimination loop (phases UNITS up to EXP in figure 7.1). Values are computed with respect to the total amount of created nodes. The second line shows the percentages of nodes which have been deleted in the course of optimizations with respect to the total amount of created nodes, and the third line is similar to the second but values are computed with respect to the total amount of deleted nodes. For example, in Nenofex (GF, RR), 67.47 % of all created nodes were deleted in the variable elimination loop, 5.89 % of all created nodes were deleted in the course of global flow and redundancy removal, which accounts for 8.34 % of all deleted nodes.

Figures 7.7 to 7.9 show plots of sorted ratios of deleted nodes in solved instances. Ratios are plotted on the vertical axis. These sorted ratios were taken to compute the values in table 7.2. In figure 7.7, a value of 1 indicates that an instance has been solved entirely by expansions without SAT solving, for values 0 the instances contained only one type of variables at start and hence SAT solving could immediately be applied.

Although the percentages of nodes deleted by optimizations are small, more instances can be solved when optimizations are enabled (see table 7.1). Global flow and redundancy removal seem to have a positive effect on the number of solved instances, even though the immediate influence on the number of deleted nodes is minor.

Tables 7.4 to 7.5 show the numbers of instances where either both or only one of two versions of Nenofex succeeded, timed out or ran out of memory. In any table, the best-performing version (GF, RR) is compared to another. The versions where optimizations are partially (table 7.4) or totally (table 7.5) switched off are, apart from few exceptions, not capable of solving instances uniquely, that is instances which version (GF, RR) could not solve .

Thus switching on both global flow and redundancy removal seems to be justified. It enables Nenofex to solve the largest number of instances compared to the three other versions, furthermore to run out of memory on the smallest number of instances, but at the same time has the drawback of causing time outs on the largest number of instances.

Concerning expansions, LCA computation and marking variables for score update, statistical observations are listed in the following. Data has been drawn from runs on instances where Nenofex (GF, RR) succeeded.

- Table 7.6 shows the relative frequencies (percentage) of the eight possible expansion cases (defined in section 4.3.3 on page 50) summed up over all instances where the three versions of Nenofex succeeded.

- *parent pointer dereferences in LCA computation:* for each instance which was solved by Nenofex (GF, RR), the average number of parent pointer dereferences per call of function *compute_lca* in algorithm 2 (page 45) has been computed. The arithmetic mean over these average values is 3.3 and the maximum is 9.3, which suggests that the NNF-tree has a flat structure.

- *variables marked for score update in innermost scope:* for each instance which was solved by Nenofex (GF, RR), the average percentage of LU-marked, ISU-marked and DSU-marked variables in the innermost scope with respect to the amount of remaining variables in that scope has been computed before each expansion (before the cheapest variable is selected for expansion, scores are recomputed for marked variables in this scope only). The arithmetic mean over these average values is roughly 4.48% for all three types of marks, hence in the majority of cases, all three marks were set. The LCA and scores of roughly 36.3 variables had to be recomputed. This value is the arithmetic mean over the average numbers of variables in the innermost scope which were either LU-marked, ISU-marked or DSU-marked before variable selection. This suggests that the marking policy for LCA and score updates is justified: the scores of only a small amount of variables in the innermost scope needs to be recomputed. In contrast, it can be expected that recomputation of scores of *all* variables in this scope takes more time than recomputation of scores of marked variables only.

		Nenofex		
	Quantor	*GF, RR*	*no GF, RR*	*no GF, no RR*
Solved	**421**	361	352	313
OOT	**32**	124	103	83
OOM	683	**651**	681	740

Table 7.1: 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).

	(GF, RR)	**(no GF, RR)**
del./created	67.47	65.90
del. by opt./created	5.89	4.27
del. by opt./deleted	8.34	6.74

Table 7.2: Influence of optimizations on the percentage of deleted nodes.

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

Table 7.3: Number of instances where both or only one of Quantor and Nenofex (GF, RR) succeeded, timed out or ran out of memory.

	(GF, RR) only	**Both**	**(no GF, RR) only**	**Sum**
Solved	14	347	5	366
OOT	37	87	16	140
OOM	14	637	44	695

Table 7.4: Number of instances where both or only one of Nenofex (GF, RR) and Nenofex (no GF, RR) succeeded, timed out or ran out of memory.

	(GF, RR) only	**Both**	**(no GF, no RR) only**	**Sum**
Solved	50	311	2	363
OOT	51	73	10	134
OOM	6	645	95	746

Table 7.5: Number of instances where both or only one of Nenofex (GF, RR) and Nenofex (no GF, no RR) succeeded, timed out or ran out of memory.

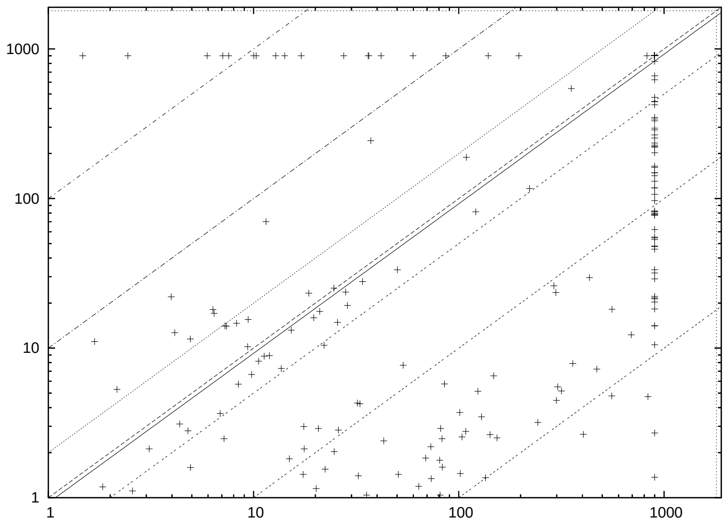


Figure 7.2: Scatter plot of penalized runtimes of Quantor and Nenofex (GF, RR). Below the diagonal Quantor is faster.

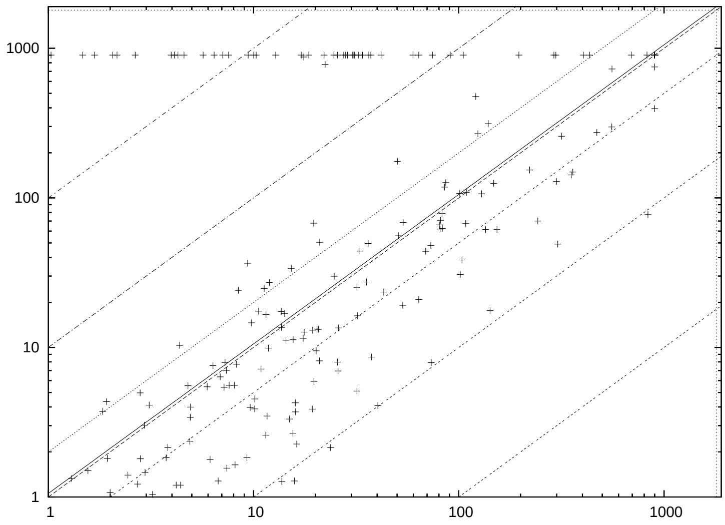


Figure 7.3: Scatter plot of penalized runtimes of Nenofex (GF, RR) and Nenofex (no GF, no RR). Below the diagonal Nenofex (no GF, no RR) is faster.

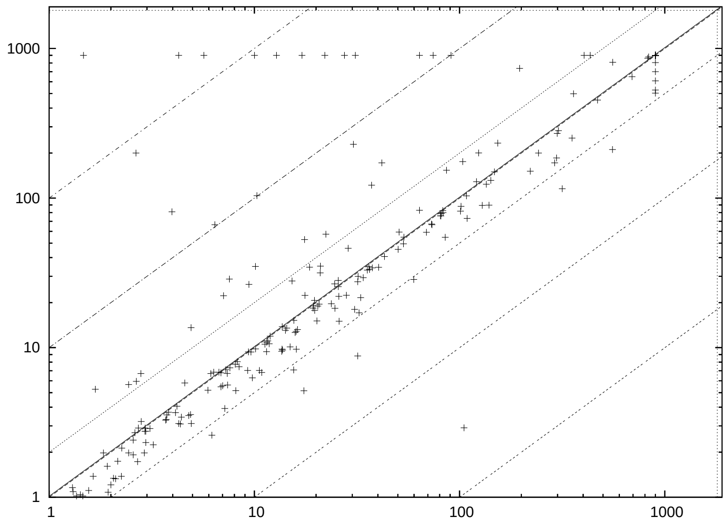


Figure 7.4: Scatter plot of penalized runtimes of Nenofex (GF, RR) and Nenofex (no GF, RR). Below the diagonal Nenofex (no GF, RR) is faster.

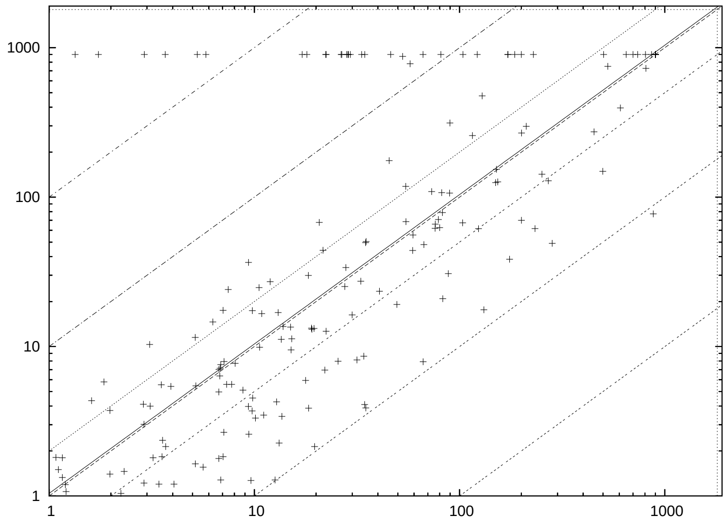


Figure 7.5: Scatter plot of penalized runtimes of Nenofex (no GF, RR) and Nenofex (no GF, no RR). Below the diagonal Nenofex (no GF, no RR) is faster.

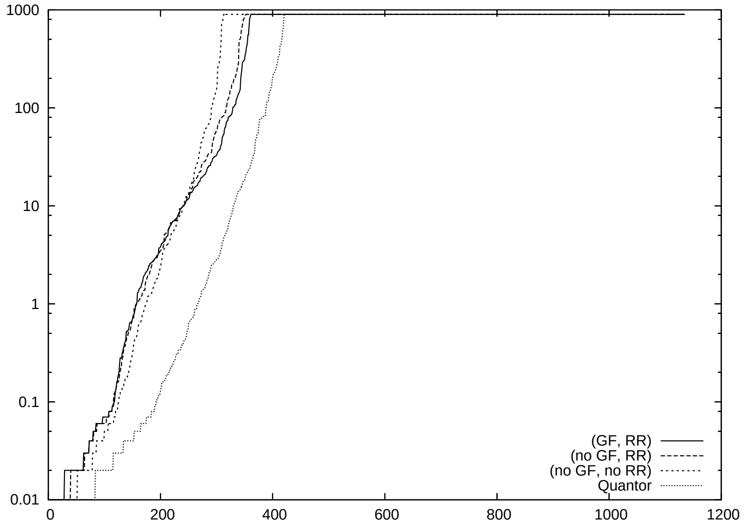


Figure 7.6: Plot (logarithmic scale on vertical axis) of sorted penalized runtimes

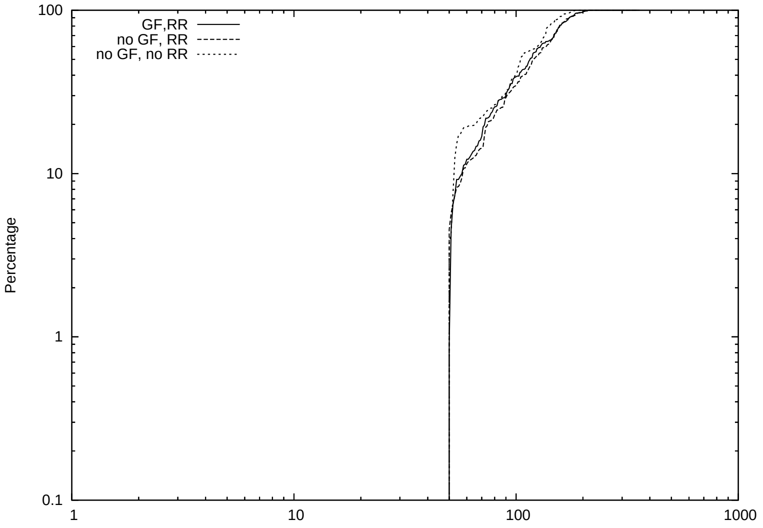


Figure 7.7: Plot (logarithmic scale on both axes) of sorted ratios of total deleted nodes with respect to total created nodes in solved instances

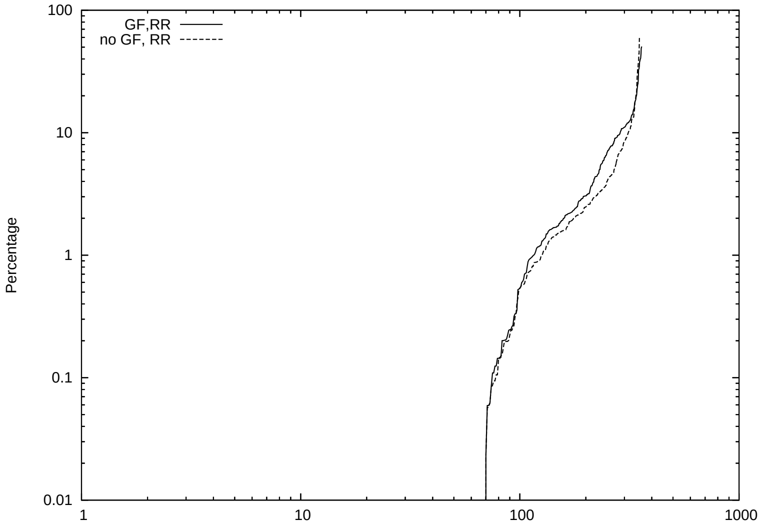


Figure 7.8: Plot (logarithmic scale on both axes) of sorted ratios of total nodes deleted by GF and RR with respect to total created nodes in solved instances

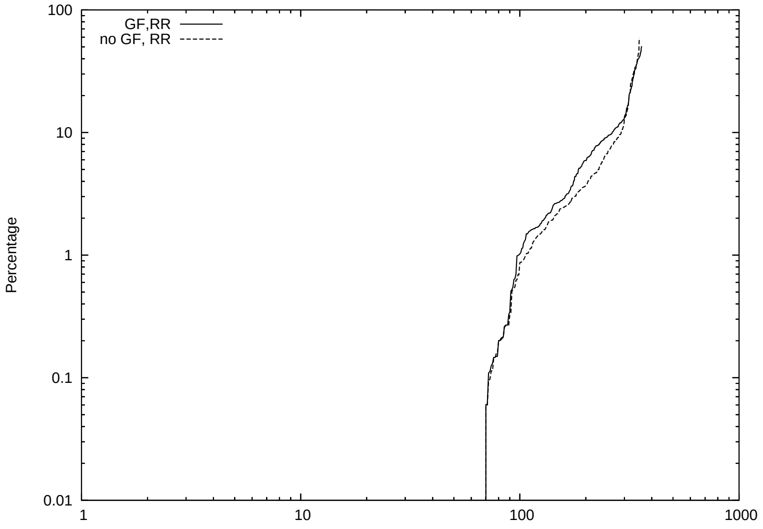


Figure 7.9: Plot (logarithmic scale on both axes) of sorted ratios of total nodes deleted by GF and RR with respect to total deleted nodes in solved instances

	(GF, RR)	(no GF, RR)	(no GF, no RR)
$\langle\exists, \vee, =\rangle$	8.65	7.47	7.16
$\langle\exists, \vee, <\rangle$	0.3	0.20	0.01
$\langle\exists, \wedge, =\rangle$	0.18	0.22	0.25
$\langle\exists, \wedge, <\rangle$	90.65	91.9	92.32
$\langle\forall, \vee, =\rangle$	0.00	0.00	0.00
$\langle\forall, \vee, <\rangle$	0.10	0.10	0.13
$\langle\forall, \wedge, =\rangle$	0.02	0.02	0.03
$\langle\forall, \wedge, <\rangle$	0.10	0.09	0.10

Table 7.6: Relative frequencies (percentage) of expansion cases

Chapter 8

Summary

In this report Nenofex, an expansion-based solver for quantified boolean formulae (QBF) has been presented. The solver works on negation normal form (NNF) and successively eliminates variables from the two innermost scopes by expansion. The motivation for the use of NNF instead of conjunctive normal form (CNF) as formula representation relies on the observation that expansion of some existential variable on NNF is involved with linear size increase of the formula, whereas eliminating the same variable by resolution on CNF can increase the formula quadratically in the worst case.

In Nenofex, a formula in NNF is represented as an NNF-tree, which is a structurally restricted tree consisting of operator nodes (logical conjunction and disjunction) and positive or negative literal nodes. An operator node may have an arbitrary number of children, yet must have at least two. Thus operator nodes denote an n-ary boolean function. Nodes which have only one child left are merged with their parent. The type of child nodes must be different from the type of their parent. This restriction corresponds to applications of the law of associativity of logical conjunction and disjunction. Operator nodes must not have multiple or complementary literal nodes of one and the same variable as children. These situations are avoided by applying one-level simplification. The structural restrictions must be preserved under any modifications of the NNF-tree, which does not require fully traversing the tree but can be carried out locally at the regions where modifications took place. The purpose of structural restrictions is to keep the size of the NNF-tree small and the distance between nodes and the root short, which was partially confirmed by experimental results.

Expanding a variable in Nenofex is performed locally by copying relevant parts of the NNF-tree only. In order to expand some variable, only the expansion-relevant subtree is copied. This is the smallest subtree which contains all occurrences of the variable. Thus copying irrelevant subtrees is avoided within the expansion procedure. Expansion in Nenofex relies on an extended notion of least common ancestors (LCAs) of variables. The LCA of two nodes in the NNF-tree is the one common ancestor of the nodes which is farthest away from the root. The LCA of a variables is the LCA over all of its occurrences. The expansion-relevant LCA of a variable is an extension of its LCA which takes into account the set of all children of the LCA whose subtrees contain at least one occurrence of that variable. This definition establishes a correspondence between expansion-relevant subtrees and expansion-relevant LCAs of variables. In the algorithm for incremental computation of expansion-relevant LCAs, which follows from the definition, an explicit upward-directed search is carried out starting from each occurrence of the variable. Concerning runtime, the algorithm will profit from flat NNF-trees. When the expansion-relevant subtree of a variable is copied, node properties are updated in interleaved fashion. Thus the NNF-tree need not be fully traversed to carry out this task.

An expansion is involved with a potential increase of the size of the NNF-tree. In Nenofex expansions are scheduled in order to keep the size small. For this purpose, always the cheapest variable with respect to a cost measure is expanded. The cost measure for expansions is the size increase in terms of number of nodes that is caused when a variable is expanded. Variables are assessed based on their score, which is a pessimistic estimate on the actual expansion costs. Score computation is based on expansion-relevant LCAs and is carried out by considering the number of nodes which are added to (increase score) and deleted from (decrease score) the NNF-tree in an expansion. Increase scores are exact and can be computed efficiently. In contrast, decrease scores are not exact. These are computed by anticipating the immediate effects of variable assignments during expansion. Variables are virtually assigned true and false and the sizes of virtually deleted subtrees (that is the numbers of nodes in the subtrees) are summed up. Nodes which would be virtually deleted by one-level simplification are not counted which renders decrease scores inaccurate but allows simpler computation. The scores of variables will change if the NNF-tree is modified. Instead of recomputing the scores of all variables, variables whose LCA or scores have been affected by modifications are marked for score update. Affected variables are identified by inspecting parts of the NNF-tree where modifications occur. Experiments show that the scores of only a small amount of variables had to be recomputed on average after an expansion, which justifies the use of update marks.

Redundancy is removed from the NNF-tree by means of two approaches from circuit optimization. In global flow, implications are derived from signals which are then used for transforming the circuit to reduce its size. ATPG-based redundancy removal relies on the detection of untestable faults in the circuit. A fault which can not be tested does not affect the circuit's function and hence the corresponding hardware can be removed. In Nenofex, the implementation of global flow and ATPG-based redundancy removal is very limited and incomplete. Not all redundant parts in the NNF-tree can be detected. Despite the limitations, experiments suggest that these two approaches are crucial for achieving best performance in terms of the number of solved instances, but also that they are involved with considerable runtime overhead.

Variables are expanded in cyclic fashion in Nenofex. After units and unates have been eliminated until saturation, redundancy is removed in a small part of the NNF-tree until a cutoff criterion is satisfied. The cheapest variable from the two innermost scopes is expanded. If the expansion of an existential variable causes the size increase to exceed a heuristic threshold, then a universal expansion is forced and the threshold is increased. Scores are updated for variables from the innermost scope only. If there is only one type of variables left in the formula, then a CNF will be generated from the NNF-tree which is forwarded to a SAT solver. This way, the SAT solver performs a check for tautology or satisfiability of the formula represented by the NNF-tree.

Experiments confirm the hypothesis that existential expansion on NNF yields smaller formulae than resolution on CNF. In a comparison with Quantor, a CNF-based solver with similar strategy, Nenofex ran out of memory on less instances, but also had more time outs. With respect to scheduling heuristics and runtime improvements, there is still room for optimizations.

Appendix A

Figures and Algorithms

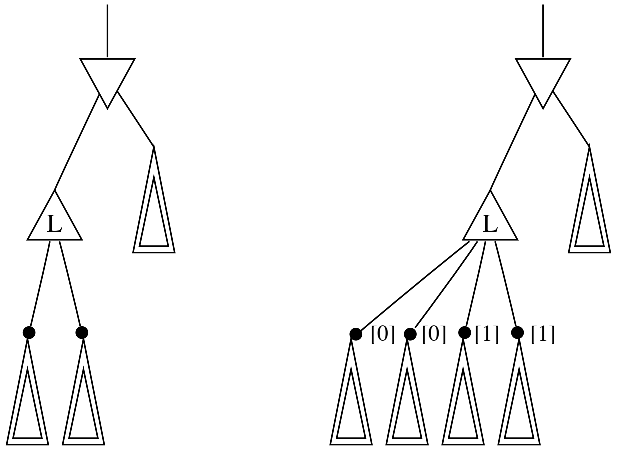


Figure A.1: Expansion template for case $\langle\forall, \wedge, =\rangle$

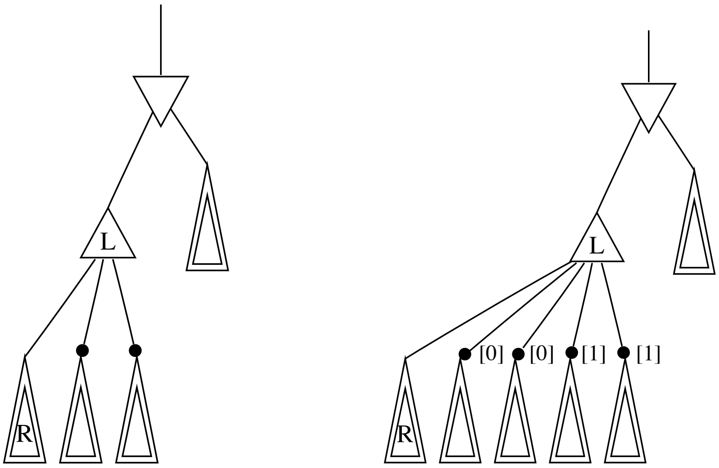


Figure A.2: Expansion template for case $\langle \forall, \wedge, < \rangle$

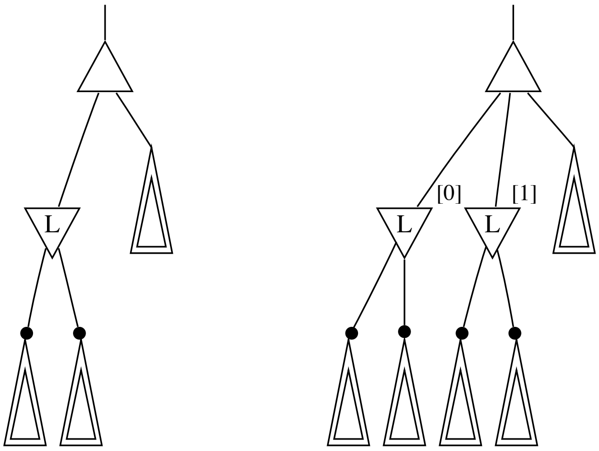


Figure A.3: Expansion template for case $\langle \forall, \vee, = \rangle$

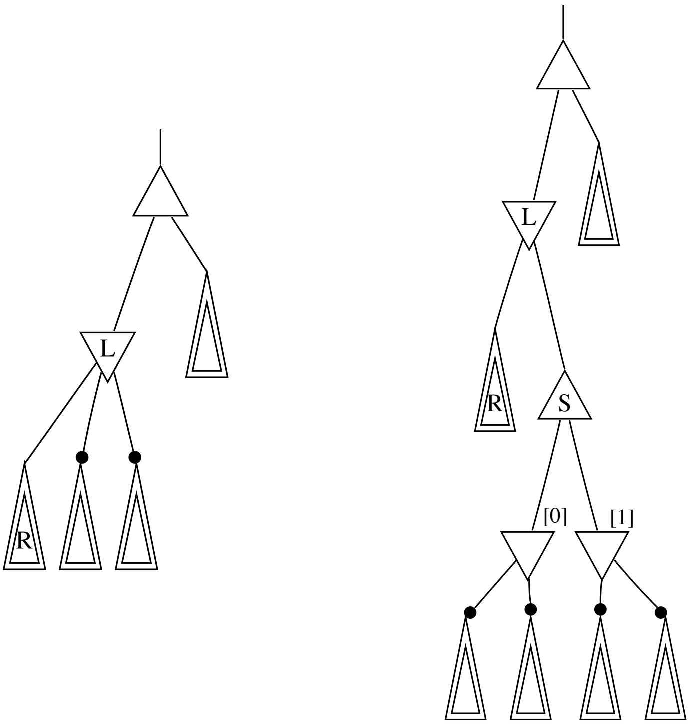


Figure A.4: Expansion template for case $\langle \forall, \vee, < \rangle$

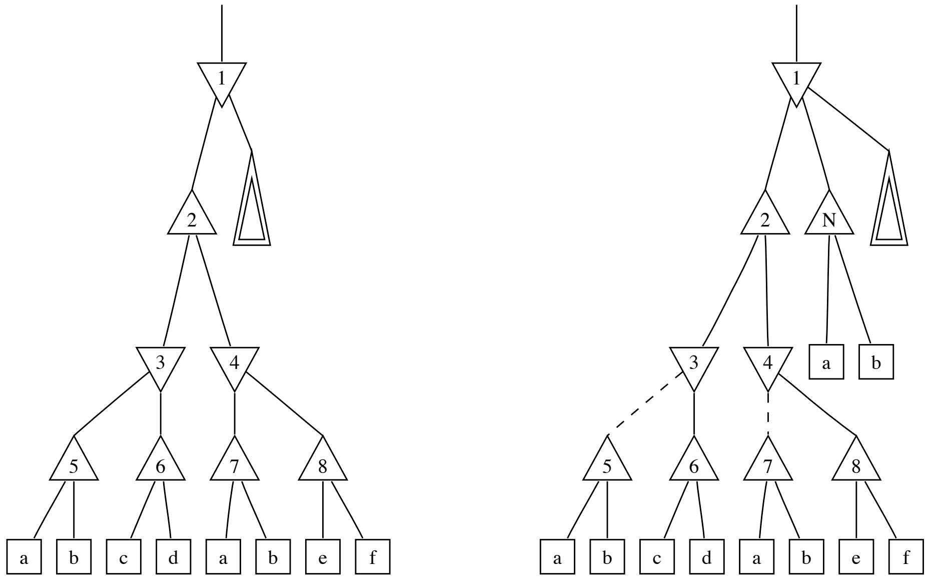


Figure A.5: Transformation related to operator nodes as implicants

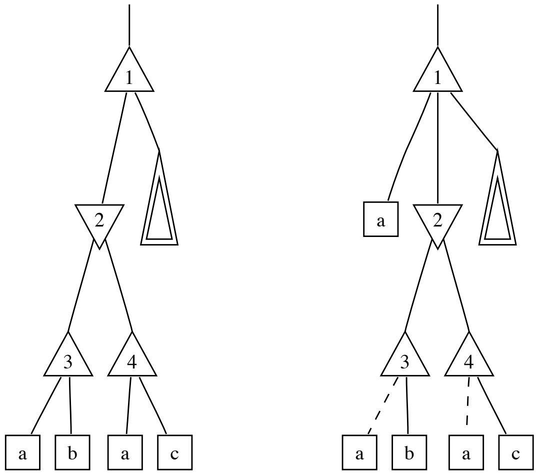


Figure A.6: Transformation related to literal nodes as implicants

Algorithm 12: forward_propagate_truth

```
Input: node n to be assigned true
Result: further propagations (backward or forward) or conflict (flag conflict = 1)
Data: nodes n, ch, p and pp, global flag conflict

assert(conflict = 0)
assert(n unassigned)
assign_false(n), n.justified ← 1
p ← parent(n)
pp ← parent(p)
if p is AND then
    assert((p is unassigned) or (p is false))
    if (p is false and not justified) or
      (p unassigned and not on fault path and pp on fault path) then
        if p has exactly one unassigned child then
            ch ← find_unassigned_child(p)
            backward_propagate_falsity(ch)
            if p unassigned and conflict = 0 then
                forward_propagate_falsity(p)
            p.justified ← 1
    else if p unassigned then
        if all children of p assigned then
            assert(all children assigned true)
            forward_propagate_truth(p)
else
    assert(p is OR)
    if n not on fault path and p on fault path then
        conflict ← 1
    else if p not assigned then
        forward_propagate_truth(p)
    else
        assert(p is true)
        p.justified ← 1
```

Algorithm 13: backward_propagate_falsity

Input: node n to be assigned false
Result: further propagations (backward) or conflict (flag conflict = 1)
Data: node n, variable var, global flag conflict, queue propagation_queue

```
assert(conflict = 0)
assert(n unassigned)
if n is a literal then
    var ← variable(n)
    if var unassigned then
        if n negated then assign_true(var) else assign_false(var)
        enqueue(propagation_queue, var)
    else
        if (n negated and var false) or (n not negated and var true) then
            conflict ← 1
else if n is OR then
    assign_false(n)
    forall children ch of n and conflict = 0 do
        assert(ch is literal or AND)
        if ch unassigned then
            backward_propagate_falsity(ch)
    n.justified ← 1
else
    assert(n is AND)
    assign_false(n)
    if n has exactly one unassigned child then
        ch ← find_unassigned_child(n)
        backward_propagate_falsity(ch)
        n.justified ← 1
```

Bibliography

[AB02] A. Ayari and D. A. Basin. QUBOS: Deciding quantified boolean logic using propositional satisfiability solvers. In M. Aagaard and J. W. O'Leary, editors, *FMCAD*, volume 2517 of *Lecture Notes in Computer Science*, pages 187–201. Springer, 2002.

[BB04] P. Bjesse and A. Borälv. DAG-aware circuit compression for formal verification. In *ICCAD*, pages 42–49. IEEE Computer Society / ACM, 2004.

[BB06] R. Brummayer and A. Biere. Local two-level and-inverter graph minimization without blowup. *Proc. 2nd Doctoral Workshop on Mathematical and Engineering Methods in Computer Science (MEMICS'06)*, October 2006.

[BB07] U. Bubeck and H. Kleine Büning. Bounded universal expansion for preprocessing QBF. In Marques-Silva and Sakallah [MSS07], pages 244–257.

[BCCZ99] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In R. Cleaveland, editor, *TACAS*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer, 1999.

[BCM+92] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Inf. Comput.*, 98(2):142–170, 1992.

[Ben05a] M. Benedetti. Quantifier trees for QBFs. In F. Bacchus and T. Walsh, editors, *SAT*, volume 3569 of *Lecture Notes in Computer Science*, pages 378–385. Springer, 2005.

[Ben05b] M. Benedetti. skizzo: A suite to evaluate and certify QBFs. In R. Nieuwenhuis, editor, *CADE*, volume 3632 of *Lecture Notes in Computer Science*, pages 369–376. Springer, 2005.

[Bie04] A. Biere. Resolve and expand. In H. H. Hoos and D. G. Mitchell, editors, *SAT (Selected Papers*, volume 3542 of *Lecture Notes in Computer Science*, pages 59–70. Springer, 2004.

[BKF95] H. Kleine Büning, M. Karpinski, and A. Flögel. Resolution for quantified boolean formulas. *Inf. Comput.*, 117(1):12–18, 1995.

[BL94] H. Kleine Büning and T. Lettmann. *Aussagenlogik: Deduktion und Algorithmen*. B.G.Teubner Stuttgart, 1994.

[Bry86] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.

[CE81] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In D. Kozen, editor, *Logic of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer, 1981.

[CGS98] M. Cadoli, A. Giovanardi, and M. Schaerf. An algorithm to evaluate quantified boolean formulae. In *AAAI/IAAI*, pages 262–267, 1998.

[DIM93] DIMACS. Satisfiability Suggested Format, 1993.

[DLL62] M. Davis, G. Logemann, and D. W. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962.

[dlT92] T. Boy de la Tour. An optimality result for clause form translation. *J. Symb. Comput.*, 14(4):283–302, 1992.

[DP60] M. Davis and H. Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, 1960.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[GNT01a] E. Giunchiglia, M. Narizzano, and A. Tacchella. Backjumping for quantified boolean logic satisfiability. In B. Nebel, editor, *IJCAI*, pages 275–281. Morgan Kaufmann, 2001.

[GNT01b] E. Giunchiglia, M. Narizzano, and A. Tacchella. Quantified Boolean Formulas satisfiability library (QBFLIB), 2001. `www.qbflib.org`.

[HR04] M. Huth and M. Ryan. *Logic in Computer Science. Modelling and Reasoning about Systems*. Cambridge University Press, 2004.

[HS04] H. H. Hoos and T. Stützle. *Stochastic Local Search: Foundations & Applications*. Elsevier / Morgan Kaufmann, 2004.

[JB07] T. Jussila and A. Biere. Compressing BMC encodings with QBF. *Electr. Notes Theor. Comput. Sci.*, 174(3):45–56, 2007.

[JBS$^+$07] T. Jussila, A. Biere, C. Sinz, D. Kröning, and C. M. Wintersteiger. A first step towards a unified proof checker for QBF. In Marques-Silva and Sakallah [MSS07], pages 201–214.

[KPKG02] A. Kuehlmann, V. Paruthi, F. Krohm, and M. K. Ganai. Robust boolean reasoning for equivalence checking and functional property verification. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 21(12):1377–1394, 2002.

[KS97] W. Kunz and D. Stoffel. *Reasoning in Boolean Networks: Logic Synthesis and Verification Using Testing Techniques*. Kluwer Academic Publishers, Norwell, MA, USA, 1997. Foreword By R. E. Bryant.

[MLB00] V. D. Agrawal M. L. Bushnell. *Essentials of Electronic Testing for Digital, Memory and Mixed-Signal VLSI Circuits*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[MSS07] J. Marques-Silva and K. A. Sakallah, editors. *Theory and Applications of Satisfiability Testing - SAT 2007, 10th International Conference, Lisbon, Portugal, May 28-31, 2007, Proceedings*, volume 4501 of *Lecture Notes in Computer Science*. Springer, 2007.

[OW02] T. Ottmann and P. Widmayer. *Algorithmen und Datenstrukturen, 4. Auflage*. Spektrum Akad. Verl., 2002.

[PG86] D. A. Plaisted and S. Greenbaum. A structure-preserving clause form translation. *J. Symb. Comput.*, 2(3):293–304, 1986.

[QDI05] QDIMACS standard, version 1.1, 2005. `http://www.qbflib.org/qdimacs.html`.

[Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, 1965.

[SAT07] SAT Competition, 2007. `http://www.satcompetition.org/2007/`.

[SC85] A. Prasad Sistla and E. M. Clarke. The complexity of propositional linear temporal logics. *J. ACM*, 32(3):733–749, 1985.

[SS96] J. P. Marques Silva and K. A. Sakallah. GRASP - a new search algorithm for satisfiability. In *ICCAD*, pages 220–227, 1996.

[Tse68] G. S. Tseitin. On the complexity of derivation in propositional calculus. *Studies in Constructive Mathematics and Mathematical Logic*, 1968.

[ZM02a] L. Zhang and S. Malik. Conflict driven learning in a quantified boolean satisfiability solver. In L. T. Pileggi and A. Kuehlmann, editors, *ICCAD*, pages 442–449. ACM, 2002.

[ZM02b] L. Zhang and S. Malik. The quest for efficient boolean satisfiability solvers. In *CAV '02: Proceedings of the 14th International Conference on Computer Aided Verification*, pages 17–36, London, UK, 2002. Springer-Verlag.

Lebenslauf

Persönliches

Name:	Florian Matthias Lonsing
Geburtsdatum/-ort:	12. April 1983 in Linz

Ausbildung

Okt. 2005 –	Magisterstudium Informatik, Johannes Kepler Universität Linz, voraussichtlich bis Jänner 2008
Okt. 2002 – Okt. 2005	Bakkalaureatsstudium Informatik, Johannes Kepler Universität Linz
Okt. 2001 – Apr. 2002	Grundwehrdienst
Sep. 1993 – Jun. 2001	Allgemeinbildende Höhere Schule (Gymnasium) in Linz
Sep. 1989 – Jul. 1993	Volksschule in Linz

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Magisterarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Florian Lonsing   Linz, Dezember 2007