FAKULTÄT FÜR !NFORMATIK

Extending a Tableau-based SAT Procedure with Techniques from CNF-based SAT

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Computational Intelligence

ausgeführt von

Leopold Carl Robert Haller
Matrikelnummer 0355898

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung:
Betreuer: Ao. Univ.-Prof. Dr. Uwe Egly

Wien, 1.12.2008

(Unterschrift Verfasser) (Unterschrift Betreuer)

Abstract

Das propositionale Erfüllbarkeitsproblem (SAT) ist ein klassisches Entscheidungsproblem der theoretischen Informatik. Es war das erste Entscheidungsproblem, für das NP-Vollständigkeit bewiesen wurde, und Implementierungen von Lösungsalgorithmen für SAT-Instanzen werden seit den frühen 1960ern erforscht. Ein spezieller Fokus liegt hier historisch auf der Betrachtung von SAT-Problemen, die in konjunktiver Normalform (KNF) gegeben sind.

In den letzten fünfzehn Jahren hat sich die Effizienz solcher KNF-basierter Lösungsalgorithmen enorm verbessert. Programme, die das Erfüllbarkeitsproblem lösen (sogenannte SAT-Solver), finden heute vielfältige Anwendung in Industrie und Forschung, etwa in Soft- und Hardwareverifikation und in Logik-basierter Planung.

In vielen praktischen Anwendungsgebieten von SAT-Solvern sind die Eingabedaten als strukturierte Formeln oder Schaltkreise gegeben. Um solche Instanzen mit KNF-basierten Solvern zu lösen muss zuerst ein Übersetzungsschritt durchgeführt werden. Dabei geht die ursprüngliche Strukturinformation der Formel verloren, wodurch der Einsatz strukturbasierter Heuristiken zum Beschleunigen des Lösungsprozesses unmöglich wird.

In dieser Arbeit wird eine Erweiterung des BC-Tableaukalküls zur Feststellung der Erfüllbarkeit von beschränkten kombinatorischen Schaltkreisen vorgestellt. Eine kurze Einführung in propositionale Logik und das Erfüllbarkeitsproblem (SAT) wird gegeben, und es wird der klassische Davis-Logemann-Loveland-Algorithmus (DLL) zur Lösung von SAT-Instanzen in konjunktiver Normalform präsentiert. Es wird aufgezeigt, wie moderne KNF-Solver das grundlegende DLL-Schema um nicht-chronologisches Backtracking und Lernen erweitern. Techniken werden beschrieben, mithilfe derer SAT-Solver in der Lage sind praktisch relevante Probleme in Industrie und Forschung zu lösen, und es werden Ansätze diskutiert um das Lösen des SAT-Problems in Schaltkreisinstanzen zu beschleunigen.

Es wird gezeigt, dass eine Implementierung des BC-Tableaus als generalisierte DLL-Prozedur gesehen werden kann, und es wird dargestellt, wie sich Techniken aus dem Bereich des KNF-basierten SAT-Solving in das BC-Tableau integrieren lassen. Ein Prototyp einer solchen erweiterten Tableauprozedur wurde entwickelt und seine Effektivität im Vergleich zu bestehenden SAT-Solvern evaluiert. Es zeigt sich, dass die Erweiterung des BC.Tableaus die durchschnittliche Geschwindigkeit in Benchmarks wesentlich verbessert, und dass das erweiterte BC-Tableau als Grundlage für Schaltkreis-basierte SAT-Solver durchaus mit modernen KNF-basierten Implementierungen Schritt halten kann.

Abstract

The propositional satisfiability problem (SAT) is one of the central decision problems in theoretical computer science. It was the first decision problem that was proven to be NP complete, and the study of implementations of decision procedures for SAT date back to the early 1960s. In the area of satisfiability research, work on SAT instances given in conjunctive normal form (CNF) has been a major focus of research.

In the last 15 years, the efficacy of CNF-based SAT algorithms, i.e., algorithms for instances of propositional formulas in conjunctive normal form (CNF), has increased significantly. Today, SAT solvers are employed in a number of applications in industry and science such as software and hardware verification or logic-based planning.

In many application areas of SAT, the instances are originally given as structured formulas or circuit instances. Using a CNF-based SAT solver on such instances requires a translation step from the original formula to CNF. The result lacks the structural information of the original instance, which could have been used heuristically to speed up the solving process.

In this thesis, we present an extension of the BC tableau calculus for determining satisfiability of constrained Boolean circuits. We give a short introduction to propositional logic and the SAT problem, and we present classical algorithms for solving SAT such as the Davis-Putnam (DP) procedure and the Davis-Logemann-Loveland (DLL) procedure. Modern extensions to the basic DLL framework, such as non-chronological backtracking and clause learning, are discussed, which reduce solving time on industrial instances considerably. We also present some approaches for solving the SAT problem in circuits.

We show that a BC-tableau-based SAT algorithm can be seen as a generalization of the basic DLL procedure and how techniques from CNF-based SAT can be integrated into such a tableau procedure. We present a prototypical implementation of these ideas and evaluate it using a set of benchmark instances. The extensions increase the efficiency of the basic BC tableau considerably, and the framework of our extended BC-tableau solver is shown to be competitive with state-of-the-art CNF-based solving procedures.

Contents

Chapter 1

Introduction

The propositional satisfiability problem (SAT) is the problem of deciding for a propositional formula whether there is a variable assignment under which the formula evaluates to true. In computer science, the SAT problem has a history as a problem of both theoretical and practical interest. Early implementations of solvers, such as the Davis-Logemann-Loveland procedure, date back to the early 1960s [12; 13]. The SAT problem was also the first decision problem proved to be NP complete in the early 1970s [10].

More recently, work on the propositional satisfiability problem has shifted somewhat towards the practical side. SAT solvers have been applied to a number of real-world problems, including hardware and software verification and logic-based planning. At the same time, building efficient solvers for the propositional satisfiability problem has become a major focus of research.

An enormouse effort so far has concentrated on extending the basic Davis-Logemann-Loveland procedure which works on satisfiability instances given in conjunctive normal form (CNF), but, increasingly, this work is being extended to non-clausal SAT instances. Intuitively, it seems likely that the added structural information can be used to speed up the solving process, but work on non-clausal SAT still has a long way to go. At this time, state-of-the-art CNF-based SAT solvers coupled with translation front-ends for non-clausal instances still outperform dedicated non-clausal solvers [45].

One of the reasons for this is that a lot of effort has been invested into engineering efficient CNF-based solvers. The structurally simple CNF format lends itself very well to fast and cache-efficient low-level implementation techniques and can be naturally extended with techniques such as clause-learning and non-chronological backtracking. Circuit-based solvers have yet to match this level of engineering.

In this thesis, we will give a brief introduction to propositional logic, the propositional satisfiability problem, and its applications. We will explain some techniques from the area of CNF-based SAT which have proven to increase efficiency in practical instances, and we will show some of the attempts that have been made to solve the SAT problem directly in structured problem instances. Finally, we will present the BC tableau, a tableau calculus for solving the SAT problem in Boolean circuits, and we will discuss how a BC tableau procedure can be extended in a generalized DLL framework with many of the techniques used in CNF-based SAT, such as non-chronological backtracking, learning, and restarts. We present BattleAx^3 ("BattleAx Cube" being an anagram of "BC Tableau Ext."), a prototype of such an extended BC-based procedure, describe its implementation, and compare it both to the original BC procedure and to MiniSAT, an efficient CNF-based solver.

The structure of this thesis is as follows. Chapter 2 introduces basic terminology and notation and gives a brief introduction to propositional logic. It also presents the original Davis-Putnam (DP) [12] and Davis-Logemann-Loveland (DLL) [13] procedures that form the basics of modern solvers. Chapter 3 gives an overview of modern CNF-based SAT solving. We explain extensions to the basic DLL framework such as non-chronological backtracking and learning, efficient implementation techniques, and common solving heuristics. In Chapter 4, the area of solving SAT for circuit instances is discussed. This includes a description of circuit-to-CNF translation techniques, circuit extensions for CNF-based SAT solvers, and dedicated circuit-based solvers. Chapter 5 presents the BC tableau and shows how a BC-based solver can be implemented in a generalized DLL framework and extended with many of the techniques found in CNF-based SAT. It also gives a detailed description of the prototypical implementation BattleAx^3. Chapter 6 provides benchmarking results for BattleAx^3, compares a number of different solving strategies, and compares it with the original BC procedure and the CNF-based MiniSAT SAT solver. Finally, Chapter 7 provides a conclusion.

Chapter 2

The Propositional Satisfiability Problem

The propositional satisfiability problem is a central problem in computer science from a theoretical as well as from a practical point of view. It has historical importance as the first problem that was proven to be NP-complete [10], and its analysis and the development of satisfiability decision procedures have spawned a vast array of literature.

One of the main reasons for the high interest in the satisfiability problem is that implementations of satisfiability decision procedures, so-called SAT solvers, have a wide range of possible applications, many of them industrial rather than academic in nature. Advances in SAT-solving from the last fifteen years have made it possible to go beyond toy instances and solve propositional encodings of real-world problems from various domains, such as logic-based planning, automated test pattern generation, and software and hardware verification.

This chapter will serve as a short introduction to propositional logic and the propositional SAT problem. The DP algorithm and its successor, the DLL algorithm, will be described, the latter being the algorithmic framework that still forms the basis of most modern SAT solvers.

2.1 Basic Definitions and Terminology

The work in this thesis uses two closely related notions to describe problem instances of satisfiability, propositional logic and Boolean circuits. In this section, we introduce basic terminology and notation and formally characterize both of them.

2.1.1 Propositional Logic

Propositional logic (also called propositional calculus, sentential logic, or combinatorial logic) is a branch of mathematical logic that deals with the analysis of well-formed logical formulas built up from propositional atoms and logical connectives.

First, we introduce the syntax of propositional logic by giving an inductive definition of the set $\mathcal{F}$ of propositional formulas.

Definition 1. *Let $\mathcal{B}$ be a countable set of Boolean variables. Then the set $\mathcal{F}$ of all well-formed propositional formulas is defined inductively as follows.*

(i) $\mathcal{B} \cup \{\top, \bot\} \subset \mathcal{F}$.

(ii) If $\phi \in \mathcal{F}$, then $(\neg\phi) \in \mathcal{F}$.

(iii) If $\phi, \psi \in \mathcal{F}$, then $(\phi \circ \psi) \in \mathcal{F}$ for $\circ \in \{\wedge$(conjunction)$, \vee$(disjunction)$, \Rightarrow$ (implication)$, \Leftrightarrow$ (equivalence)$\}$.

For easier readability, brackets may be omitted. In this case, we define the order of binding strength (from strongest to weakest) to be $\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow$.

A formula $v \in \mathcal{B}$ is called a *propositional atom*. An atom or a negated atom is called a *literal*. Such a literal is said to be in positive or negative phase depending on whether its variable is negated. An unnegated variable is said to be in *positive phase*, while a negated variable is said to be in *negative phase*. The opposite phase literal to a literal l is referred to as $\bar{l}$. A disjunction of literals is called a *clause*.

Definition 2. *Let $\mathcal{V} = \{\mathbf{T}, \mathbf{F}\}$ be the set of* Boolean values*. A function $f : \mathcal{V}^n \mapsto \mathcal{V}$ is called a* Boolean function *of arity n.*

Atoms represent simple statements which can assume the Boolean values true ($\mathbf{T}$) or false ($\mathbf{F}$) when modeling some arbitrary domain with propositional logic. Atoms may represent any such proposition, such as "it is raining" or "the value of variable x is greater than zero". Non-atomic formulas describe compound statements whose truth value is related to the component truth values by a Boolean formula. A statement

$$\text{“it is raining”} \Rightarrow \text{“Anne will stay at home”}$$

for example, is false only if it is raining and Anne leaves her home and true otherwise. This relationship does not depend on the semantic contents of the propositions that are associated with the atoms, but only on their truth values. In order to be able to determine the truth value of such arbitrary statements mechanically, we need to introduce the formal semantics of propositional logic. The first step is to introduce a formal device that assigns truth values to atomic propositions.

Definition 3. *For a set of Boolean variables $\mathcal{P}$, a (possibly partial) function $I : \mathcal{P} \mapsto \mathcal{V}$ is called an interpretation.*

We will call a function I interpretation of a formula ϕ if I is an interpretation of the Boolean variables occurring in ϕ. If I is a partial function, we refer to it as a partial interpretation. Interpretations will also be referred to as variable assignments. An interpretation I' is an extension of an interpretation I, or more concisely, $I \subseteq I'$, iff

$$\text{for any } x, \text{ if } I(x) \text{ is defined, then } I(x) = I'(x)$$

Since we want to be able to evaluate the truth values of arbitrary formulas, we extend I to formulas in the following way.

Definition 4. *For a given interpretation I', let $I : \mathcal{F} \mapsto \mathcal{V}$ be its extension to formulas, defined in the following way:*

(i) $I(\top) = \mathbf{T}$

(ii) $I(\bot) = \mathbf{F}$

(iii) $I(v) = I'(v)$ **iff** $v \in \mathcal{B}$

(iv) $I(\neg\phi) = \mathbf{T}$ **iff** $I(\phi) = \mathbf{F}$

(v) $I(\phi \wedge \psi) = \mathbf{T}$ **iff** $I(\phi) = I(\psi) = \mathbf{T}$

(vi) $I(\phi \vee \psi) = \mathbf{T}$ **iff** $I(\phi) = 1$ *or* $I(\psi) = \mathbf{T}$

(vii) $I(\phi \Rightarrow \psi) = \mathbf{T}$ **iff** $I(\phi) = \mathbf{F}$ *or* $I(\psi) = \mathbf{T}$

(viii) $I(\phi \Leftrightarrow \psi) = \mathbf{T}$ **iff** $I(\phi) = I(\psi) = \mathbf{T}$ *or* $I(\phi) = I(\psi) = \mathbf{F}$

For easier readability, no distinction will be made between interpretations and their extensions to formulas. Furthermore, if it is clear from the context which interpretation I is being referred to, we will use the notation $v := X$ to indicate that $I(v) = X$. We now introduce some further notions that will be needed later on.

Definition 5. *An interpretation I of a propositional formula ϕ is called a* model *of ϕ* **iff** *$I(\phi) = \mathbf{T}$. The set of all models of ϕ is* $\mathrm{Mod}(\phi)$.

For example, given the formula $a \Rightarrow b$, both $\{a \mapsto \mathbf{F}, b \mapsto \mathbf{T}\}$ and $\{a \mapsto \mathbf{T}, b \mapsto \mathbf{F}\}$ are interpretations, but only the former is a model. A number of equivalent ways will be used to express the model attribute of an interpretation.

Remark. *Given a propositional formula ϕ, the following notions are equivalent:*

(i) *I is a model of ϕ*

(ii) $I \models \phi$

(iii) *I satisfies ϕ*

(iv) $I \in \mathrm{Mod}(\phi)$

Definition 6. *A formula ϕ is* satisfiable **iff** $\mathrm{Mod}(\phi) \neq \emptyset$*, otherwise it is* unsatisfiable.

Definition 7. *A formula ϕ is* valid **iff** *for every interpretation I, it holds that $I \models \phi$.*

We now define the subformula relations on formulas.

Definition 8. *The formula ϕ is an* immediate subformula *of ψ* **iff** *one of the following conditions holds*

(i) $\psi = \neg\phi,$

(ii) $\psi = (\phi \circ \omega)$ *or* $\psi = (\omega \circ \phi)$ *for* $\circ \in \{\wedge, \vee, \Rightarrow, \Leftrightarrow\}$.

The formula ϕ is a subformula *of ψ* **iff** *(ϕ, ψ) is in the transitive closure of the immediate subformula relation.*

2.1.2 Boolean Circuits

Boolean circuits are mathematical models of digital combinatorial circuits, i.e., digital circuits where no outputs are fed back into the circuit as inputs. They are closely related to propositional formulas. Translation strategies will be described in Chapter 4.

The following characterization of Boolean circuits is mostly taken from Drechsler, Juntilla, and Niemelä [14].

Definition 9. *A* Boolean circuit $\mathcal{C}$ *is a pair* $(\mathcal{G}, \mathcal{E})$*, where*

(i) $\mathcal{G}$ *is a non-empty finite set of* gates

(ii) $\mathcal{E}$ *is a set of equations where*

- *each equation is of the form* $g := f_g(g_1, \ldots, g_n)$*, where* $g, g_1, \ldots g_n \in \mathcal{G}$ *and* $f_g : \mathcal{V}^n \mapsto \mathcal{V}$ *is a Boolean function,*
- *each gate* $g \in G$ *appears at most once on the left-hand side in the equations in* $\mathcal{E}$*, and*
- *the dependency graph,* $\mathrm{graph}(\mathcal{C}) = (\mathcal{G}, \{(g, g') \mid g := f(\ldots, g', \ldots)\})$ *is acyclic, i.e., no gate is defined recursively.*

A gate that does not occur on the left hand side of any equation in $\mathcal{E}$ is called a *primary input gate*, a set that does not occur on the right hand side of any equation is called a *primary output gate*. The set of primary input and primary output gates of a circuit $\mathcal{C}$ is designated by *input*$(\mathcal{C})$ and *output*$(\mathcal{C})$ respectively.

For any gates g and g', if g' appears on the right-hand side of g's equation, we call g a parent of g' and g' a child of g. The ancestor and descendant relation between gates are defined intuitively as the transitive closures of the parent and child relation respectively.

A *subcircuit* is a part of a larger circuit,

Definition 10. *Given two Boolean circuits* $C = (\mathcal{G}, \mathcal{E})$ *and* $C' = (\mathcal{G}', \mathcal{E}')$*, we call* C' *a* subcircuit *of* C *iff*

- $\mathcal{E}' \subseteq \mathcal{E}$ *and*
- $\mathcal{G}' = \{\, v, c_1, \ldots, c_n \mid v := f_v(c_1, \ldots, c_n) \in \mathcal{E}' \,\}$

In this thesis, only certain classes of Boolean functions will be associated with gates in gate equations. These include

- the constant functions *true*$() = \mathbf{T}$ and *false*$() = \mathbf{F}$,
- *not*: $\mathcal{V} \mapsto \mathcal{V}$, *not*$(\mathbf{T}) = \mathbf{F}$, *not*$(\mathbf{F}) = \mathbf{T}$
- *and*: $\mathcal{V}^n \mapsto \mathcal{V}$, *and*$(v_1, \ldots, v_n) = \mathbf{T}$ iff all v_1 to v_n are $\mathbf{T}$.
- *or*: $\mathcal{V}^n \mapsto \mathcal{V}$, *or*$(v_1, \ldots, v_n) = \mathbf{T}$ iff at least one of v_1 to v_n is $\mathbf{T}$.

- *ite*: $\mathcal{V}^3 \mapsto \mathcal{V}$, this is the if-then-else construct. The value of $ite(v_c, v_1, v_2)$ is the value of v_1 if $v_c = \mathbf{T}$ and the value of v_2 if $v_c = \mathbf{F}$.
- *odd*: $\mathcal{V}^n \mapsto \mathcal{V}$, $odd(v_1, \dots, v_n) = \mathbf{T}$ iff the number of input values v_c with $v_c = \mathbf{T}$ is odd.
- *even*: $\mathcal{V}^n \mapsto \mathcal{V}$, $even(v_1, \dots, v_n) = \mathbf{T}$ iff the number input values v_c with $v_c = \mathbf{T}$ is even.
- *equiv*: $\mathcal{V}^n \mapsto \mathcal{V}$, $equiv(v_1, \dots, v_n) = \mathbf{T}$ iff $v_1 = \dots = v_n$.
- $card_x^y$: $\mathcal{V}^n \mapsto \mathcal{V}$, this is the cardinality gate, $card_x^y(v_1, \dots, v_n) = \mathbf{T}$ iff at least x and at most y input values are $\mathbf{T}$.

We call a (possibly partial) function $\tau : \mathcal{G} \mapsto \mathcal{B}$ a truth assignment of $\mathcal{C}$. A partial truth assignment is a truth assignment whose function is partial. If for a gate g, $\tau(g)$ is defined, we say that g is *assigned.* A truth assignment in which all gates are assigned is a *total* truth assignment.

A truth assignment τ' is an extension of an assignment τ iff $\tau(g) = \tau'(g)$ for all gates $g \in \mathcal{G}$ that are assigned in τ, i.e., $\tau \subseteq \tau'$.

A total truth assignment τ is *consistent* in a circuit $\mathcal{C}$ iff, for each gate $g \in \mathcal{G}$ with associated Boolean function $f_g(g_1, \dots, g_n)$, it holds that $f_g(\tau(g_1), \dots, \tau(g_n)) = \tau(g)$. We call a gate g *justified* in a truth assignment τ if, for any satisfying truth assignment τ' where for each of g's children g_c $\tau'(g_c) = \tau(g_c)$ (if $\tau(g_c)$ is defined), it holds that $\tau(g) = \tau'(g)$. Intuitively, a gate is justified in τ if its output is fully explained by the values of its children that are set in τ. An AND-gate with value $\mathbf{F}$ in τ, for example, is justified, if one of its children also has value $\mathbf{F}$ in τ. No matter what values the other children will take on, the gate output will not change.

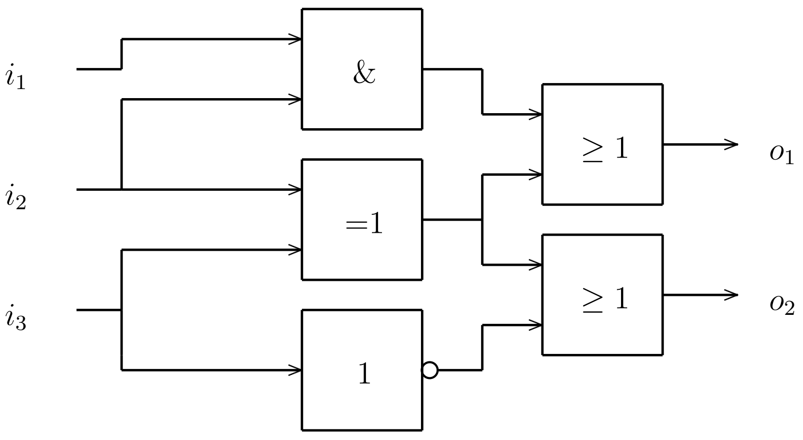
It is easy to see that a given circuit C has $2^{|input(\mathcal{C})|}$ consistent truth assignments, that is, there is one distinct satisfying truth assignment for each truth assignment on the input gates. Such an assignment can be found by determining the values of all non-input gates by applying the corresponding gate function in a bottom-up fashion.

A *constrained circuit* is a pair $(\mathcal{C}, \tau)$ where $\mathcal{C}$ is a Boolean circuit and τ is a non-empty (possibly partial) truth assignment. A constrained circuit is called satisfiable, iff there is an extension τ' of τ that is consistent in $\mathcal{C}$.

The behavior of a Boolean circuit can be modeled by a propositional formula. The Boolean function for the value of a given output gate can be determined by starting with its associated Boolean function and replacing each gate occurrence with its own function in turn, until no more replacements are possible. We will denote this exhaustive replacement process of a formula f with $expand(f)$. A full propositional modeling of a circuit $\mathcal{C}$'s behavior is then given by

$$\bigwedge_{o \in output(\mathcal{C})} o \Leftrightarrow expand(f_o)$$

A serious drawback of this translation is that the formula size may increase exponentially when transforming it into a normal form. More sophisticated translation strategies



$$\phi_{\mathcal{C}} = o_1 \Leftrightarrow ((i_1 \wedge i_2) \vee (i_2 \Leftrightarrow \neg i_3)) \wedge o_2 \Leftrightarrow ((i_2 \Leftrightarrow \neg i_3) \vee \neg i_3)$$

Figure 2.1: Example of a Boolean circuit and a corresponding propositional formula.

which circumvent this problem will be described in detail in Chapter 4. An example of a Boolean circuit and a corresponding propositional formula is shown in Figure 2.1.

2.2 The Propositional Satisfiability Problem

The propositional satisfiability problem (SAT) is a central decision problem in computer science, and it can be stated in its general form in the following way:

Definition 11. *For a given propositional formula ϕ, the* Boolean satisfiability problem *(SAT) is to decide whether ϕ is satisfiable, i.e., whether there is an interpretation I of ϕ so that $I \models \phi$.*

The propositional satisfiability problem was the first decision problem proven to be NP-complete in [10], that is, it can be solved by a non-deterministic Turing machine in polynomial time, and every other member of NP can be cast into an instance of SAT by a polynomial-time transformation algorithm. All known algorithms that decide the Boolean satisfiability problem have an exponential worst-case time complexity.

Much work on the SAT-problem has been done on propositional formulas in normal forms, the most prominent being *conjunctive normal form* (CNF). A CNF formula is a conjunction of clauses, it can be pictured as a two-level circuit where multiple OR-gates feed into one AND gate. We can easily transform any given formula ϕ into an equivalent CNF-formula by exhaustively replacing subformulas of ϕ with the substitutions from Table 2.1.

The problem with this kind of transformation is that a non-normalized propositional formula can be exponentially more succinct than its corresponding CNF. In Chapter 4,

original	**substitution**
$\neg\neg\phi$	ϕ
$\neg(\phi \vee \psi)$	$\neg\phi \wedge \neg\psi$
$\neg(\phi \wedge \psi)$	$\neg\phi \vee \neg\psi$
$\phi \Rightarrow \psi$	$\neg\phi \vee \psi$
$\phi \Leftrightarrow \psi$	$(\phi \Rightarrow \psi) \wedge (\psi \Rightarrow \phi)$
$(\phi_1 \wedge \phi_2) \vee \psi$	$(\phi_1 \vee \psi) \wedge (\phi_2 \vee \psi)$

Table 2.1: Simple transformation to CNF.

more sophisticated translation strategies will be described which avoid this problem by introducing new variables for subformulas.

For many problems, a circuit is a more direct representation of the original problem. We have already determined that a Boolean circuit C has $2^{|input(\mathcal{C})|}$ consistent truth assignments, therefore it only makes sense to determine the satisfiability of a constrained circuit. For constrained circuits, we can define the satisfiability problem in the following way.

Definition 12. *For a constrained circuit* $(\mathcal{C}, \tau)$*, the* Boolean circuit satisfiability problem *(CIRCUIT-SAT) is to decide whether* $(\mathcal{C}, \tau)$ *is satisfiable, i.e., whether there is a consistent extension of* τ*.*

A number of decision problems about propositional formulas can be cast into the form of a satisfiability problem.

Remark. *For two propositional formulas* ϕ *and* ψ *it holds that*

(i) ϕ *is valid* **iff** $\neg\phi$ *is unsatisfiable.*

(ii) $\phi \models \psi$ **iff** $\neg(\phi \Rightarrow \psi)$ *is unsatisfiable.*

(iii) ϕ *and* ψ *are equivalent* **iff** $\neg(\phi \Leftrightarrow \psi)$ *is unsatisfiable.*

Furthermore, most SAT-solvers do not simply return a binary answer to the satisfiability problem, but they also provide a model for satisfiable instances. The information encoded in this model can be used to gain more information about the problem. In SAT instances obtained from hardware or software verification problems, an interpretation can yield an error trace leading up to a problem state, in SAT-based planning, an interpretation can be transformed into a ready-made plan. A number of decision problems about propositional formulas can be cast into the form of a satisfiability problem.

For an illustration of how a given problem can be encoded into a propositional form, see the example provided in Figure 2.2.

2.3 Basic Algorithms for the SAT-Problem

Developing solvers that perform reasonably well on interesting classes of instances (such as encodings of real-world problems) is a hard problem, both from a theoretical as well

1	3		
	4		
			2

Sudoku is a simple puzzle that can be easily transformed into a SAT-instance. In the small example to the left it consists of a 2×2 arrangement of 2×2 boxes. Each small 2×2 box should be filled with the numbers from 1 to 4, likewise, each row and column should contain each of the numbers. In our encoding, we use variables of the form $n_{x,y}$, where $I(n_{x,y}) = 1$ means that the number n is at the location x, y. In order to represent the puzzle rules we need to define some propositional constraints.

First of all, an auxiliary construction is needed that builds from a set of propositional atoms S a propositional formula that is true iff exactly one of the atoms in S is true.

$$\mathit{exactlyOne}(S) = \bigvee_{q \in S} q \wedge \bigwedge_{q \in S} (\bigwedge_{r \in S \setminus \{q\}} \neg q \vee \neg r)$$

We can now define the actual constraints of the Sudoku domain.

$$\begin{aligned}
\mathit{uniqueAtLoc} &= \bigwedge_{(x,y) \in \{1,2,3,4\}^2} \mathit{exactlyOne}(\{1_{x,y}, 2_{x,y}, 3_{x,y}, 4_{x,y}\}) \\
\mathit{uniqueInRow}(n, y) &= \mathit{exactlyOne}(\{n_{1,y}, n_{2,y}, n_{3,y}, n_{4,y}\}) \\
\mathit{uniqueInCol}(n, x) &= \mathit{exactlyOne}(\{n_{x,1}, n_{x,2}, n_{x,3}, n_{x,4}\}) \\
\mathit{uniqueInBox}(n, b_x, b_y) &= \mathit{exactlyOne}(\{n_{2*b_x+\Delta_x, 2*b_y+\Delta_y} \mid (\Delta_x, \Delta_y) \in \{1,2\}^2\})
\end{aligned}$$

These constraints encode that each grid location contains exactly one number (*uniqueAtLoc*), and that each row, column, and 2×2 box contain each number from one to four exactly once (*uniqueInRow*, *uniqueInCol*, *uniqueInBox*). Now we only need to set up the initial information set in the instance shown in the picture above,

$$\mathit{init} \ = \ 1_{1,1} \wedge 3_{2,1} \wedge 4_{2,2} \wedge 2_{4,3}$$

The resulting propositional formula is now,

$$\begin{aligned}
\phi = \mathit{uniqueAtLoc} \ &\wedge \ \mathit{init} \\
&\wedge \ \bigwedge_{n \in \{1,2,3,4\}} \bigwedge_{(b_x,b_y) \in \{0,1\}^2} (\mathit{uniqueInBox}(n, b_x, b_y)) \\
&\wedge \ \bigwedge_{n \in \{1,2,3,4\}} \bigwedge_{i \in \{1,2,3,4\}} (\mathit{uniqueInRow}(n, i) \wedge \mathit{uniqueInCol}(n, i))
\end{aligned}$$

After finding an interpretation that satisfies ϕ we can simply extract a solution to the Sudoku instance by looking at the values of the variables of the form $n_{x,y}$. If there is no solution, a SAT-solver would report the instance to be unsatisfiable.

Figure 2.2: Example for SAT encoding.

as from an engineering point of view. A conceptually simple but inefficient algorithm to determine the satisfiability of a given formula ϕ, would be to enumerate all possible interpretations I and check whether any of those I satisfies ϕ. Although this algorithm shares with all other known SAT algorithms a worst-case exponential time-complexity, it is not competitive in the average case for instances occurring in practice. This enumeration procedure also makes the NP-membership of SAT intuitive: A non-deterministic Turing machine can "guess" an interpretation I and then check in polynomial time whether I satisfies ϕ.

In the literature, most algorithms are based on propositional formulas given in CNF. These can be further divided into two main classes, stochastic algorithms and systematic algorithms. The former tend to view SAT as an optimization problem with the goal of maximizing the number of satisfied clauses and employ stochastic search strategies through the space of interpretations. If such strategies are not combined with systematic approaches, stochastic procedures are incomplete, that is, they may be able to find a satisfying interpretation for a formula, but they cannot show that a given instance is unsatisfiable.

An incomplete SAT solver can still be useful. In many verification applications, for example, a satisfying interpretation yields a trace leading up to an error state. In such a case, an incomplete solver can help to find bugs, but it can never prove that the given system is bug-free.

Systematic algorithms, on the other hand, typically build partial assignments in a systematic way until a satisfying assignment is found or until the space of interpretations has been fully explored. Most systematic algorithms are based on the Davis-Logemann-Loveland (DLL) [13] procedure, which in turn is based on the Davis-Putnam (DP) procedure [12]. Since the work presented in this thesis is closely related to the DLL procedure, both of them will be described in this section.

2.3.1 The Davis-Putnam Algorithm

The DP algorithm was first presented in Davis and Putnam [12] and was originally used as part of a procedure for determining the validity of first order formulas (a problem that is, in general, undecidable). The DP algorithm works on a propositional formula in CNF and essentially combines the resolution rule with a search procedure.

Definition 13. *Given two clauses $c_1 = a_1 \vee \ldots \vee a_i \vee \ldots \vee a_l$ and $c_2 = b_1 \vee \ldots \vee b_j \vee \ldots \vee b_k$ where a_i and b_j are literals of the same variable v in opposite phases, i.e., $a_i = \overline{b_j}$, we call the clause*

$$c_3 = a_1 \vee \ldots \vee a_{i-1} \vee a_{i+1} \vee \ldots \vee a_l \vee b_1 \vee \ldots \vee b_{j-1} \vee b_{j+1} \vee \ldots \vee b_k$$

the resolvent *of c_1 and c_2 ($c_1 \otimes_v c_2$). The variable v is the variable resolved upon.*

The resolution rule states that given two clauses c_1 and c_2, we can infer any of their resolvents c_3.

Given a SAT instance as a propositional formula in CNF, the original DP procedure iteratively modifies the formula by a sequence of satisfiability-preserving steps using the following rules.

Literal elimination If a pair of one-literal clauses, $c_1 = a$ and $c_2 = \neg a$ exists where a is a propositional atom, conclude that the instance is unsatisfiable. If this is not the case, and a clause $c = l$ exists where l is a literal, remove all clauses that contain l and remove $\bar{l}$ from all remaining clauses. If the resulting formula is empty, conclude that the instance is satisfiable.

Affirmative-negative rule For every variable that occurs only in one phase as a literal in the CNF, remove all clauses containing that literal.

Elimination rule for propositional variables (originally referred to as "elimination rule for atomic formulas" in Davis and Putnam [12]). Choose a decision variable v and construct all possible resolvents upon that variable. Replace the original formula by a conjunction of the resolvents and all clauses in the original formula that do not contain a literal of v.

Upon closer analysis, it becomes clear that the first two rules are conceptually subsumed by the elimination rules if the critical variable v is chosen as decision variable. In order to show why this is true, consider the following scenario. If there is a one-literal clause $c = l$ on decision variable v, all opposite phase occurrences are automatically eliminated. If a second one-literal $c' = \bar{l}$ existed, its resolvent with c is the empty clause. The empty clause evaluates to false under any interpretation, therefore the instance is unsatisfiable.

A similar line of reasoning can be employed to show that the affirmative-negative rule is subsumed by the elimination rule. If a literal l only occurs in one phase, and its variable is chosen as elimination-rule variable, it has no possible resolvents. No additional clauses are therefore added to the formula in the elimination rule, but all clauses are removed that contain v. The result is then the elimination of all clauses that contained v.

By eliminating the first two rules, we can then gain a conceptually simpler variant of the DP algorithm.

Init Let ϕ be the input formula.

Step 1 If ϕ contains the empty clause, conclude unsatisfiability. If ϕ is the empty conjunction, conclude satisfiability.

Step 2 Let v be a variable occurring in ϕ. Let R be the set of all possible resolvent clauses upon the variable v from the clauses in ϕ.

Step 3 Let ϕ' be the conjunction of R and all clauses in ϕ that do not contain v.

Step 4 $\phi := \phi'$. Goto step 1.

While this method terminates after a linear number of such high-level steps, the formula may grow exponentially in size during the resolution step. The algorithm is therefore of limited use for practically interesting problem instances.

2.3.2 The Davis-Logemann-Loveland Algorithm

The DLL procedure (also referred to as DPLL) is a highly-efficient, backtracking-based algorithm for the SAT problem. It was presented in the 1962 paper by Davis, Logemann, and Loveland [13] as an improvement of the DP procedure, and it forms the basis of most competitive SAT solvers even today.

Originally, the DLL procedure was simply the DP procedure with the elimination rule replaced by the splitting rule due to the excessive worst-case memory consumption of the former.

Splitting rule Let v be variable occurring in ϕ. Let A be the conjunction of those clauses that contain v in positive phase, let B be the conjunction of clauses that contain v in negative phase, and let R be the conjunction of clauses where v does not occur. Create $A_{\setminus v}$ and $B_{\setminus \neg v}$ by removing all occurrences of v and $\neg v$ from the clauses in A and B respectively. Recursively determine the satisfiability of the formulas $A_{\setminus v} \wedge R$ and $B_{\setminus \neg v} \wedge R$. Conclude satisfiability if either of the formulas is satisfiable, else conclude unsatisfiability.

The splitting rule transforms the earlier resolution-based method into a backtrack search procedure. We will therefore recharacterize the DLL procedure in a slightly different way which emphasizes the idea of search and is closer in spirit with modern implementations of DLL and its variants.

We can view the DLL algorithm as a depth-first-search (DFS) procedure in the space of partial assignments where, between each step of the search, the following two deduction rules are applied to prune parts of the search space.

Pure literal rule If a literal occurs only in one phase in the set of unsatisfied clauses, it is set to be true in the current partial assignment.

Unit rule Let I be the current partial interpretation. If, in an unsatisfied clause, all but one literals evaluate to false, then the remaining literal is set to be true in I.

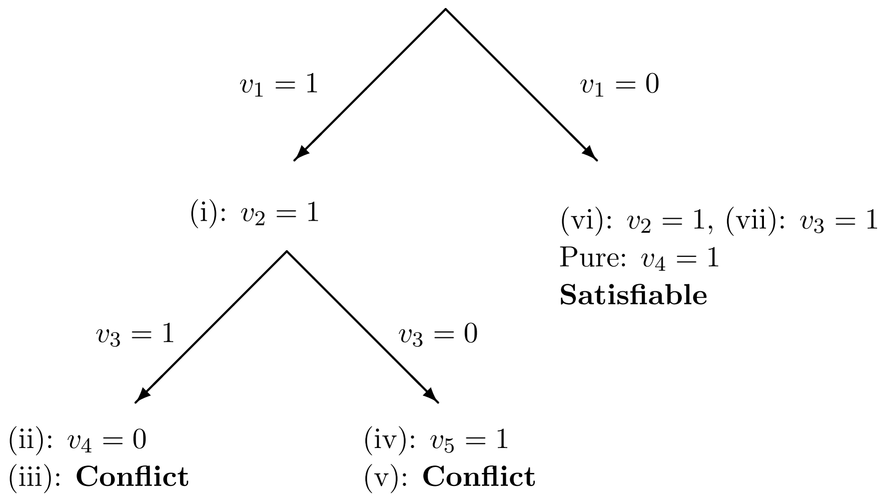
The process is initialized with the empty partial assignment and incrementally expands this assignment through search and deduction. Under the current partial assignment, a clause is called *conflicting* if it evaluates to false, *satisfied* if it evaluates to true, *unit* if all but one of its literals evaluate to false, and *unresolved* otherwise.

If a conflict occurs, that is, if a clause is conflicting under the current partial assignment, then the DFS search backtracks to an earlier node in the search tree and continues the search from there.

The simplified structure of the DLL algorithm can be seen in Algorithm 2.1; an example run is shown in Figure 2.3. The *deduce* function exhaustively applies the two deduction rules presented above. In modern implementations, the pure literal rule is

Algorithm 2.1: DLL

```
input  : $I$ - a partial interpretation, $\phi$ - a CNF formula
output: Satisfiability of $\phi$
if deduce($I, \phi$)=$Conflict$ then
    return false;
if allVarsAssigned($I, \phi$) then
    return true;
v ← chooseVar($I, \phi$);
return DLL($I \cup \{v := \mathbf{T}\}, \phi$) $\vee$ DLL($I \cup \{v := \mathbf{F}\}, \phi$)
```



$\boldsymbol{\phi}$			
(i)	**(ii)**	**(iii)**	**(iv)**
$(\neg v_1 \vee v_2)$	$(\neg v_1 \vee \neg v_2 \vee \neg v_3 \vee \neg v_4)$	$(\neg v_1 \vee \neg v_3 \vee v_4)$	$(v_3 \vee v_5)$
(v)	**(vi)**	**(vii)**	**(viii)**
$(\neg v_2 \vee v_3 \vee \neg v_5)$	$(v_1 \vee v_2)$	$(v_1 \vee \neg v_2 \vee v_3)$	$(v_4 \vee v_5)$

The above diagram depicts an example run of the DLL procedure on the input formula ϕ, with clauses numbered (i) through (viii). The tree is expanded left-to-right, applications of the unit rule and conflicts are prefixed with the numeral of the relevant clause, applications of the pure-literal rule are prefixed with "pure".

Figure 2.3: Exemplary DLL run on formula ϕ.

only used in preprocessing since the overhead costs necessary to detect the applicability of the rule outweigh the benefits of the additional deductions.

The DLL procedure is highly sensitive to the choice of decision variables. Usually, some kind of greedy heuristic is used that aims at maximizing the occurrence of possible unit-rule applications or conflicts. In most modern DLL-based solvers, heuristics are chosen which aim to produce conflicts as early as possible in order to avoid entering unnecessary regions of the search space.

2.4 Practical Applications of SAT-Solvers

Modern SAT solvers have reached a degree of efficiency where they are able to solve real-world instances with hundreds of thousands of variables. There is a multitude of possible applications, including applications in software and hardware verification, hardware testing, and logic-based planning.

In order to give some insight in how SAT solvers can be used to solve problems of practical interest, a small selection of them will be presented here.

2.4.1 Bounded Model Checking

For purposes of CNF-SAT based verification, a given Boolean circuit can be transformed into a propositional formula. Such a formula is trivially satisfiable as long as the circuit is unconstrained, i.e., as long as there is no gate that is forced to assume a specified output value. A satisfiable assignment can easily be found by assigning random values to all inputs and propagating them across the circuit. If one wants to test certain properties about the given circuit, those properties must be encoded in propositional form and added to the formula as constraints.

As an example, imagine a circuit controlling the launch of a nuclear weapon. As a security measure, two keys need to be inserted into the launching mechanism and turned at the same time in order to fire the weapon, thus requiring at least two people to initiate a launch. Let C be a Boolean circuit model of the controlling circuit. In the circuit model, the fact that a key has been turned is represented by the primary inputs k_1 and k_2 for the first and second key respectively. An assignment $k_1 := \mathbf{T}$ and $k_2 := \mathbf{T}$ would therefore represent that both keys are currently turned, and that the weapon should thus be fired. This should also be the only assignment to k_1 and k_2 that should initiate the launch sequence, which is represented itself by the primary output assignment $l := \mathbf{T}$. We could therefore introduce a design specification $(l \Rightarrow (k_1 \wedge k_2))$.

In order to verify this specification, we could translate the behaviour of the circuit into a propositional formula ϕ and check the formula $\psi = \phi \Rightarrow (l \Rightarrow (k_1 \wedge k_2))$ for validity. We can do this by using a SAT solver on the formula $\neg\psi$. If the solver returns that the formula is satisfiable, there is an error in the design of our circuit C. In the model that is returned, a state is encoded where the launch sequence is initiated $(l := \mathbf{T})$ but at least on of the keys is not turned $(k_1 := \mathbf{F}$ or $k_2 := \mathbf{F})$.

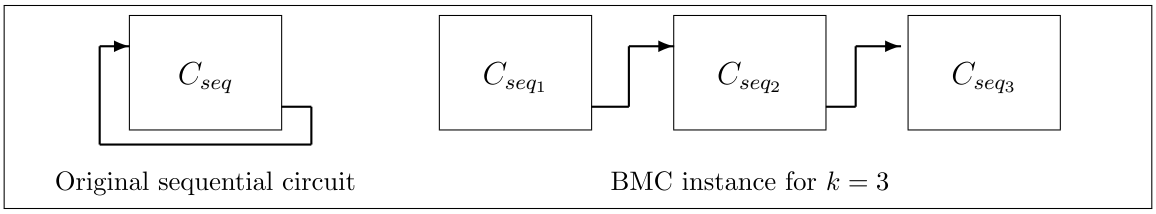


Figure 2.4: Instancing a sequential circuit for BMC.

Real problems in hardware or software verification do not come in the form of combinatorial circuits, but are usually sequential in nature. A sequential circuit cannot trivially be transformed into a propositional formula, but it is possible to construct a combinatorial circuit which simulates its behavior up to a fixed number of time-steps. The resulting circuit, together with certain properties that specify the expected behaviour can be modeled as a propositional formula and checked using a SAT solver. This technique is called *bounded model checking* (BMC) and was pioneered in Biere, Cimatti, Clarke, Fujita, and Zhu [7].

The main idea is fairly simple and will be briefly sketched here. Given a sequential circuit and a bound k, we first model the circuit as a Boolean circuit C_{seq} by removing all feedback loops. Then, k instances C_{seq_1} to C_{seq_k} of C_{seq} are created where in each, the gates are replaced by a fresh set of identical gates. Finally, for each instance, each primary output gate that feeds back into the circuit in the original sequential circuit is connected to the input gate of the next instance, e.g., the outputs of C_{seq_1} are connected to C_{seq_2}, etc. Figure 2.4 provides an example.

The resulting Boolean circuit is then translated into a propositional formula. Specifications given in some formal language such as temporal logic are translated into propositional form as well, and the conjunction of the specification and the circuit description is evaluated using a SAT solver.

By running multiple iterations of this process with increasing values of k, counterexamples to the specification with minimal length can be found. This iterative procedure is only complete if it is run until k exceeds a certain *completeness threshold* that depends on the BMC instance. If the minimal counterexample that is needed to produce the error is longer than the highest bound k that is tested, it is possible for errors to remain undetected.

2.4.2 Automatic Test Pattern Generation

During the production of microchips, certain imprecisions in the production process can lead to faulty circuits. In order to ensure the correct functioning of a chip, it is necessary to apply test input patterns to the circuit and compare them with the expected outputs. Since exhaustively testing the circuit is usually intractable, a certain set of test patterns have to be selected. This selection process, if it is to ensure the correct functioning of the chip, is not trivial. Faulty outputs of gates in the circuit may be masked by other values under certain conditions, or a gate could never be accessed in a way that produces a faulty output.

The basic problem in automatic test pattern generation (ATPG) is finding test inputs which cause output values to diverge if a specified defect exists in the circuit.

In order to be able to detect a certain class of fault, a fault model is needed first, that is, a mathematical description of the fault. One of the most commonly used models is the stuck-at-fault model (SAFM). It assumes that a gate, instead of calculating the appropriate Boolean function, is stuck at a constant truth value. In the stuck-at-fault-model, the ATPG problem for a gate g is to find input patterns that cause the output values of a circuit to diverge from the original circuit's behavior if g's output is a constant true or false signal.

While many dedicated algorithms have been developed to solve this problem, encoding the problem as an instance of SAT has proven to be very efficient given the speed of modern solvers. In a 1996 paper, Stephan, Brayton, and Sangiovanni-Vincentelli [41] already show a SAT solver to be a very robust alternative to dedicated algorithms. Since then, SAT-solver performance has increased considerably.

The basic idea for a SAT-based encoding is sketched in Figure 2.5. A circuit is constructed which encodes relevant parts of the original as well as the faulty circuit, both being connected to the same inputs. A checker is introduced, which compares the output of the two subcircuits and reports divergences. This circuit is encoded into a propositional formula, and a constraint is added that forces the checker to be true, i.e., the compound circuits' output to diverge. If a satisfying assignment is found for this circuit (which should be the case if the faulty gate is not redundant), the input values can be extracted from the satisfying interpretation and used as a test pattern.

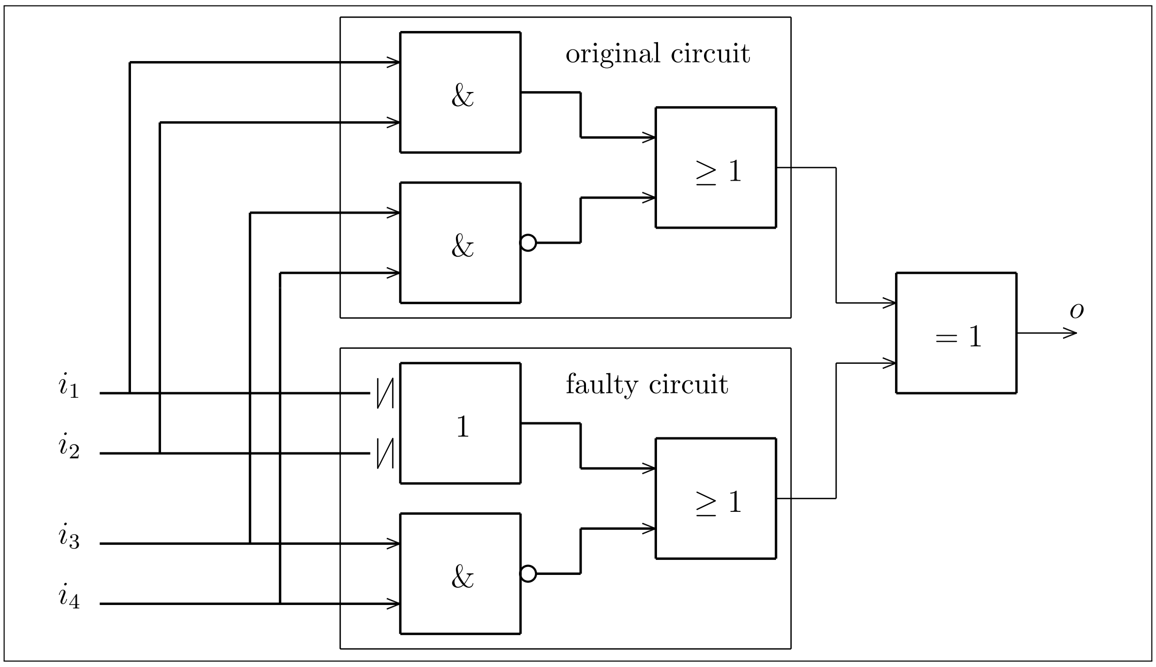


Figure 2.5: Circuit construction for ATPG-to-SAT transformation. A stuck-at-true fault for an AND-gate is analyzed. The primary output o is true if i_1 to i_4 are set the values that make the output of the original and faulty circuit diverge.

Chapter 3

Improvements Over The Standard DLL Framework

While the DLL procedure has proven to be a highly efficient framework for the development of SAT solvers, a naive implementation is unlikely to be competitive on any realistic set of benchmarks. The combination of non-chronological backtracking and learning with the DLL procedure (independently introduced in the solver Grasp [33] and in the work of Bayardo and Schrag [2]) in the mid-1990s increased the interest in the DLL procedure as a framework for SAT solvers considerably. Since then, numerous improvements and refinements have been proposed for the standard algorithm which increase its speed considerably. These can be grouped into two main categories.

First, datastructures and implementation techniques have been proposed which, while not changing the high-level behaviour of the procedure, allow for considerable speed-ups. For SAT solvers, low-level efficiency is extremely important. In modern implementations, about 90% of the runtime is spent in *Boolean Constraint Propagation* (BCP), i.e., in the exhaustive application of the unit rule. Low-level improvements can therefore easily lead to big overall speed-ups. CNF is a structurally very simple format for propositional formulas and lends itself especially well to efficient implementation techniques.

Second, improvements have been made to the way the DLL procedure searches the space of partial assignments. These include the development of various heuristics which aim at finding good decision variables and the analysis of conflicts. The information gained from a conflict can be used to prune unnecessary parts of the search space and learn deduction shortcuts that can be applied if a similar region of the search space is reentered at a later time.

In this section, low-level as well as high-level improvements to the DLL procedure will be presented that have been shown to work well on practical problem instances.

3.1 The DLL Algorithm Revisited

A recursive version of the DLL algorithm was already presented in Chapter 2. In Algorithm 3.1, a basic iterative version of the DLL algorithm is shown. This formulation is closer to actual implementations, and will be used in order to fix some basic definitions and notation.

First, a check is performed whether a conflict occurs without making any decisions, and if that is the case, the algorithm returns the instance to be unsatisfiable. Otherwise, we enter the main loop of the procedure.

First, the decide function is called, which either makes a decision and returns true, or returns false if no more unassigned variables exist. In the latter case, the instance is satisfied and the algorithm stops. Otherwise, the decision assignment is propagated via BCP in the deduce function. If a conflict occurs, the procedure backtracks to the last level where the decision variable has not been tried out in both phases. If no such level exists, that is, if all branches have been explored and found to be conflicting, the algorithm returns the instance to be unsatisfiable. After backtracking, the solver enters an unexplored branch of the search tree by flipping the previous decision assignment on the backtracking level (`flipLastDecision`).

Algorithm 3.1: Iterative DLL

```
input  : A CNF formula $\phi$
output: Satisfiability of $\phi$

$I \leftarrow \emptyset$;
dLevel $\leftarrow 0$;
if deduce($I, \phi$) = Conflict then
    return unsat;
while true do
    // Check if all variables are assigned, else make decision assignment
    if $\neg$decide($I, \phi$) then
        return sat;
    dLevel $\leftarrow$ dLevel $+ 1$;
    while deduce($I, \phi$) = Conflict do
        repeat
            undoAssignments($I$, dLevel);
            dLevel $\leftarrow$ dLevel $- 1$;
        until dLevel $> 0 \wedge$ triedBothPhases(dLevel) ;
        if dLevel = 0 then
            return unsat;
        else
            dLevel $\leftarrow$ dLevel $+ 1$;
            flipLastDecision(dLevel);
```

The variable dLevel denotes the current decision level of the solver. The decision level equals the number of decisions made on the current search branch. The decision level of a variable v, dLevel(v), is the decision level of the solver when the assignment was made. The assignment of a variable v to a value $x \in \{\mathbf{T}, \mathbf{F}\}$ in the current partial interpretation will be denoted by $v := x$. When necessary, this notation is extended to $v := x@d$ where $d = \text{dLevel}(v)$.

Note, that only the unit-rule is used in the deduce step. The pure-literal rule is usually too expensive to be applied in every step of the search process, and its elimination does not affect completeness.

3.2 Clause Learning and Non-Chronological Backtracking

In the mid-nineties, two improvements, *non-chronological backtracking* and *clause learning* were first used in SAT solving [33; 2]. In clause learning, new clauses are added to the clause database when a conflict is encountered. Such new clauses serve as deduction shortcuts if a similar region of the search space is entered later on. Non-chronological backtracking uses conflict analysis to identify possibilities for jumping back past unexplored branches, if those branches of the search tree are sure to lead to conflicts.

The combination of these techniques increased solving efficiency considerably. The general framework of the DLL algorithm with clause learning and non-chronological backtracking has been the foundation for the most efficient solvers for industrial instances to date.

3.2.1 Clause Learning

The original DLL procedure is simply a systematic search process through the space of partial assignments. Assignments are iteratively enlarged until either a satisfying assignment is found or a conflict is encountered, which causes the algorithm to backtrack. Many of the conflicts that occur during the search procedure may have the same or similar causes. Consider as an example the following CNF formula ϕ,

$$\phi = (x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \neg x_3) \wedge (\ldots \vee \ldots \vee \ldots) \wedge \ldots \wedge (\ldots \vee \ldots \vee \ldots).$$

If x_1 and x_2 are both false at some point during the search, a conflict will occur since we can deduce that $x_3 = \mathbf{T}$ and that $x_3 = \mathbf{F}$ using the unit rule. This conflict may occur exponentially often in the number of variables of ϕ. To a human, it would become clear soon that x_1 must be true whenever x_2 is false, and vice versa. This information can be used to avoid entering regions of the search space that are sure to lead to conflicts.

It is surprisingly easy to add this technique to a DLL framework. In our case, we can simply use the propositional constraint $\neg(\neg x_1 \wedge \neg x_2)$ to encode our knowledge that not both x_1 or x_2 may be false. Pushing the negation down to the literals with De Morgan's rule leaves us with the clause $(x_1 \vee x_2)$ which subsumes both of the original clauses and can be added to our original formula. Note, that this would also be the resolvent of the two clauses.

Whenever one of the two variables now becomes false, the unit rule can be immediately used to determine that the other variable must be true. We therefore save a decision and the additional inspection of the search tree that it would entail. Such a clause is called a *learned clause* or a *conflict clause* (a *conflicting clause*, in contrast, is a clause that evaluates to false under the current partial interpretation). The learned clause is redundant, the new formula with the added learned clause is therefore logically equivalent to the original one. One can think of the learned clause as a deduction shortcut since the information it encodes is already contained in the original CNF formula.

The main idea behind conflict analysis and clause learning is to generalize the above idea. Information is extracted from conflicts which is then used to avoid entering similar regions of the search space later on. This is done by tracing back a conflict to a *responsible variable assignment.*

Given a CNF formula ϕ, We call an assignment A responsible for a conflict, if A is sufficient for producing the conflict via BCP. More formally, an assignment A is responsible for an extension A' of A iff A' can be produced by repeatedly extending A using the unit rule. We call A responsible for a conflict, if it is responsible for a total assignment that is not a model.

After identifying such a responsible assignment, we can add a clause which prevents any extension of that assignment from occurring again. Intuitively, we try to find and encode the reasons for a given conflict in order to avoid running into the same conflict again later on.

In general, such an assignment can be found by repeatedly resolving the conflicting clause with those clauses that led up to the conflict by inducing assignments in BCP. Instead of using such a resolution-based characterization, we will use the notion of an *implication graph.*

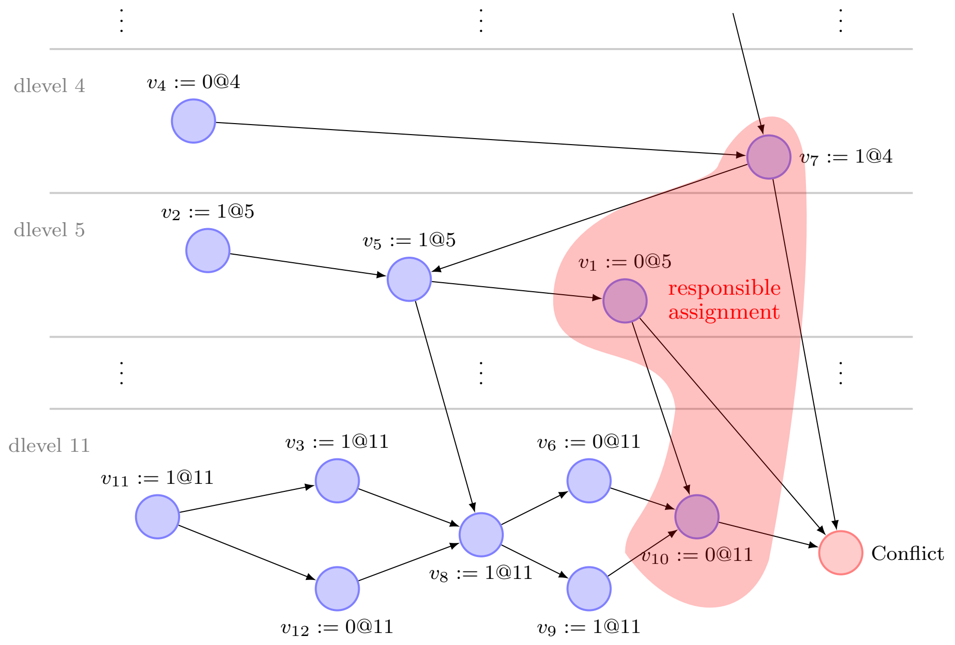
Definition 14. *At a given step of the DLL procedure on CNF formula ϕ, the* implication graph *is a directed acyclic graph $(\mathcal{A}, E)$ where*

- $\mathcal{A} = \mathcal{A}' \cup \Box$ *and $\mathcal{A}'$ is the set of all possible variable assignments, i.e.,* $\mathcal{A}' = \{\, (v := X) \mid v \in \mathrm{vars}(\phi), X \in \{\mathbf{T}, \mathbf{F}\} \,\}$ *with* $\mathrm{vars}(\phi)$ *being the set of Boolean variables occurring in ϕ. The box $\Box$ is a special conflict node.*

- *There is an edge $(a_k, a_l) \in E$* iff

  - *a_l was implied in the current DLL search branch during BCP by a clause c which contains the literal that is made to evaluate to false by a_k or*
  - *$a_l = \Box$ and a_k makes a literal in the conflicting clause evaluate to false.*

When a conflict occurs, we can traverse the implication graph $(\mathcal{A}, E)$ backwards from the conflict node in order to identify a responsible assignments. This can be done by choosing a set of vertices $S \subseteq V$, so that each path on the implication graph from a decision to the current conflict passes through at least one node in S. The set of predecessors of the conflict node, for example, constitutes a trivial responsible assignment for the conflict. By replacing nodes with their predecessors (if such predecessors exist), we



Above, an example of an implication graph at a conflict is shown. The chosen responsible assignment is indicated by the red area, the resulting conflict clause is $c_{\text{confl}} = \neg v_7 \lor v_1 \lor v_{10}$.

Newer implementations of conflict-driven backtracking (red) differ slightly from Grasp's original backtracking strategy [33] (green). Grasp backtracks to the highest decision level, newer solvers to the second-highest level.

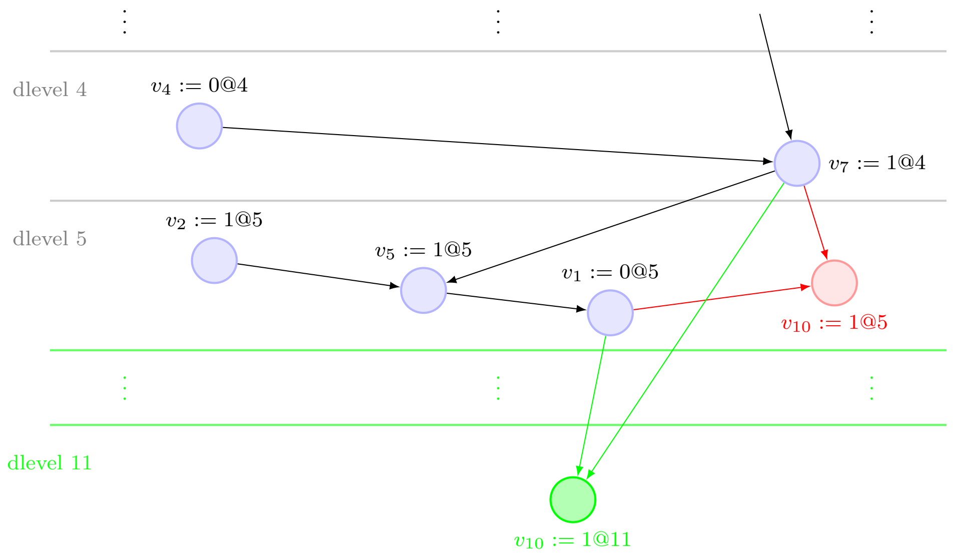


Figure 3.1: Example for learning, non-chronological backtracking, and conflict-driven assignments.

can generate other responsible assignments. This "replacement" is actually a resolution step, as was briefly described before, the variable resolved upon being the variable that was assigned a value due to BCP. For an example of a responsible assignment after a conflict, consider Figure 3.1.

We can then create a clause which acts as a constraint, ensuring that the same set of assignments will not occur again. For the set of responsible assignments S, the added conflict clause c_S contains exactly those literals which are contradicted by the assignments in S, e.g., for $S = \{x_1 := \top, x_2 := \bot, x_3 = \top\}$ we would have $c_S = \neg x_1 \vee x_2 \vee \neg x_3$. The next time a similar region of the search space is encountered, the added clause may act as an implication shortcut which may instantly force a variable assignment via BCP, where otherwise, the solver would have to resort to decisions to determine the value of the variable.

3.2.2 Non-Chronological Backtracking

After a conflict has been identified, traditional implementations of the DLL procedure would backtrack chronologically, i.e., they would reset the solver to the highest decision level where both values of a decision variable have not been tried out. The solver would then proceed the solving process by assigning to this variable the so far untried value. This technique is called chronological backtracking.

In non-chronological backtracking (also referred to as *conflict directed backjumping*), a solver may backtrack further than that, essentially leaving a number of branches unexplored. The main idea is that in backtracking, a SAT solver can skip over all unexplored branches which are sure not to lead to a satisfiable assignment. The identification of the backtracking level relies on the implication graph described above and is closely connected with the idea of learned clauses.

Non-chronological backtracking was not invented in SAT solving, but is a well studied technique originating in the area of constraint satisfaction problems under the name of "dependency directed backtracking" [40].

When a conflict is encountered, a clause is learned by analyzing the conflict and identifying a responsible assignment S from the implication graph. From this responsible assignment, we can determine the maximal decision level associated with any individual assignment in S

$$d_{\max} = \max_{(v:=X) \in S} \text{dLevel}(v).$$

Since the learned clause is added to the instance, resulting in a logically equivalent instance, the solver state will remain conflicting until at least one of the assignments in S is undone. Even if the learned clause is not added to the clause database, the solver is guaranteed to encounter only conflicts if we backtrack to any decision level $d \geq d_{\max}$ since the learned clause just provides a shortcut to an exploration of a certain part of the search space in the original instance. The unexplored branches from the current level up to the start of $d_{\max}$ can safely be skipped.

Each of the skipped branches could theoretically unfold to a partial search tree that is exponential in the size of the remaining variables. Non-chronological backtracking can result in considerable performance improvements because it prevents the solver from entering or staying in such uninteresting regions of the search space.

We can also think of non-chronological backtracking as a way of recovering the solver from the negative consequences of bad decision orderings. In a good variable ordering, conflicts are produced as soon as possible, i.e., conflicting decisions are grouped close together. In the case of a bad ordering in the traditional DLL procedure, the variable assignments for a given conflict may occur on widely separated decision levels, which may lead to an exponential increase in runtime compared to a good ordering. Non-chronological backtracking can help to identify this wide level-span between conflicting variables and jump back to the earlier levels sooner.

3.2.3 DLL with Clause Learning and Non-Chronological Backtracking

Since clause learning and non-chronological backtracking change the structure of the DLL procedure considerably, it is useful to recharacterize DLL in the iterative formulation in Algorithm 3.2. The `deduce` function performs BCP and determines if a conflict has occurred, `decide` performs a decision assignment or returns false if no more decisions can be made. The `analyze` function is called when a conflict has occurred. It adds a conflict clause to the clause database and determines a backtracking level. If it would be necessary to undo assignments at decision level 0, that is, if the conflict does not depend on any decisions made, the instance is unsatisfiable. Otherwise, backtracking is performed and the newly learned clause is used in BCP to advance into new parts of the search space.

The code in Figure 3.2 is very close to how the DLL procedure is actually implemented. Since the structure of the algorithm is somewhat complex, a more abstract, graphical representation is also given in Figure 3.2.

3.2.4 Backtracking to the Second-Highest Decision Level

Grasp's backtracking strategy is rather complex. There, the backtracking level is the most recent decision level found in the chosen responsible assignment. The first conflict of a decision always contains a literal of the most-recent decision level in any responsible assignment. Therefore, Grasp always backtracks chronologically after the first conflict that is encountered right after a decision. After this step, the first assignment that is made is the result of a learned clause, i.e., its node in the implication graph has antecedent vertices from earlier decision levels. If, at this point, a second conflict occurs, a responsible assignment can be found that contains only assignments made on earlier decision levels (by traversing the implication graph backwards past the initial assignment of the most recent decision level).

In contrast, modern solvers backtrack up to the second-highest decision level encountered in the conflict clause that is, they backtrack as far as it is possible for the conflict clause to stay an *asserting clause*, i.e., a clause where all but one literal evaluate to false under the current partial assignment. The assignments made on the second-highest decision level are not undone. Now, upon resuming BCP, the asserting clause becomes unit

Algorithm 3.2: DLL with clause learning and non-chronological backtracking.

```
input  : $\phi$ - a CNF formula
output: Satisfiability of $\phi$

$I \leftarrow \emptyset$;
if deduce($I, \phi$) = Conflict then
    return unsat;
while true do
    // Check if all variables are assigned, else make decision assignment
    if $\neg$decide($I, \phi$) then
        return sat;
    while deduce($I, \phi$) = Conflict do
        (backtrackLevel, conflictClause) $\leftarrow$ analyze($I, \phi$);
        $\phi \leftarrow \phi \cup$ conflictClause;
        if backtrackLevel $< 0$ then
            return unsat;
        else
            backtrack($I$, backtrackLevel);
```

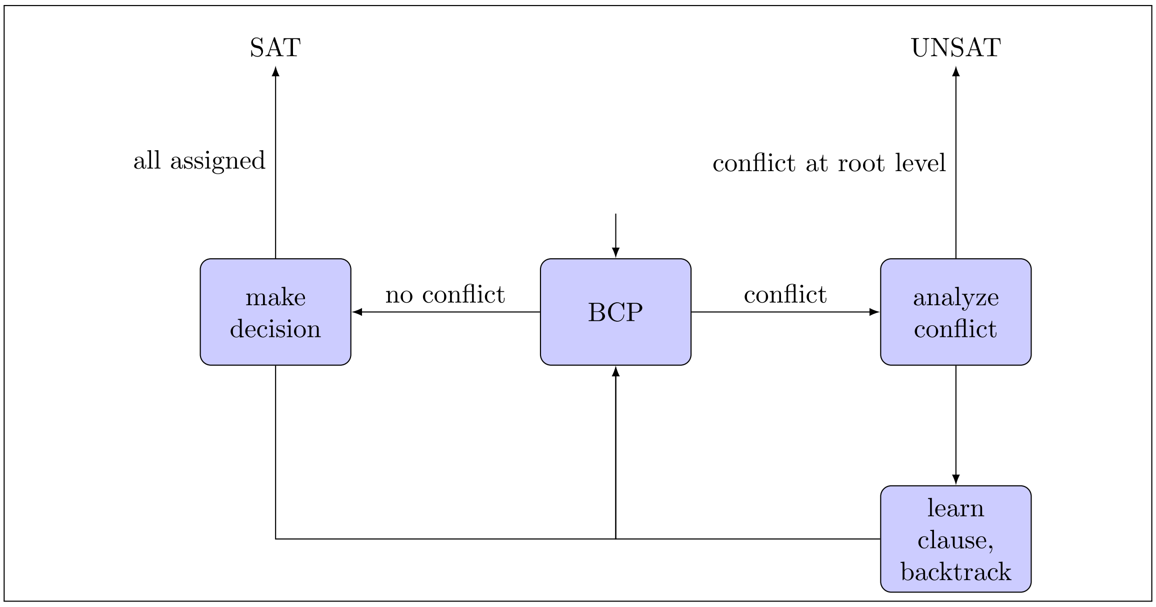


Figure 3.2: The DLL algorithm with learning and backtracking.

and the resulting conflict-driven assignment takes the SAT solver into an unexplored region of the search space. In Figure 3.1, an illustration of the two different approaches is presented via the implication graph.

The branches that are skipped between the highest and second-highest decision level of the clause are not guaranteed to lead to conflicts. Essentially, part of the viable search space is thrown away and has to be explored again. The reasons for this counter-intuitive strategy seem to be twofold, one systematic, the other technical in nature.

First, being able to deduce more at an earlier decision level may allow the solver to make heuristically smarter decisions on the direction of the search. The solver may otherwise end up in a region of the search space which—while viable in the sense that a satisfying assignment may be found—it would not have chosen heuristically given the information learned in that region. Second, jumping back to the second-highest decision level is easier to implement for a number of reasons. Together with the First-UIP clause-learning strategy (to be discussed in Section 3.3), it always produces asserting clauses. Also, in Grasp's strategy, conflict clauses may be used in BCP only some search levels after they become eligible for BCP. Consider a responsible assignment $S = \{v_1 := 0@3, v_2 := 1@4, v_3 := 1@7\}$. Grasp would backtrack to decision level 7 and assign $v_3 := 0$ via BCP, while, in fact, it could have already been assigned at decision level 4. When subsequently backtracking to level 5, for example, care should be taken that $v_3 := 0$ is not undone along with the other assignments.

Ideally, a SAT solver will choose a clause learning strategy which produces conflict clauses that are asserting clauses. This has the advantage that upon backtracking, the conflict clause becomes a unit clause and causes the solver to enter another part of the search space. This new variable assignment is then called *conflict-driven assignment.*

Conflict-driven backtracking raises the issue of completeness. If parts of the viable search space are thrown away, can we be sure that the solver does not run into a cycle where a number of partial assignments are repeated? Indeed, we can; as long as the solver produces only asserting clauses, we can prove that a solver can never run into a partial interpretation twice. First, we need to introduce some concepts that allow us to formalize parts of the DLL procedure:

For a SAT instance ϕ, we define the state of a DLL solver that follows the framework of Algorithm 3.2 as the number of times the deduce-function was called. A state S is later than a state S' if $S' < S$, and earlier if $S < S'$. The current partial interpretation for a state S is denoted by $I(S)$, the current decision level for a state is denoted by dLevel(S). A state S is non-conflicting if no clause is false under $I(S)$.

Theorem 3.2.1. *Given a DLL solver that follows the framework of Algorithm 3.2, produces only asserting conflict clauses, and backtracks to the second-highest decision level of a responsible assignment. Then there can be no pair of distinct non-conflicting states (S_1, S_2) where S_2 is later than S_1 and $I(S_1) = I(S_2)$.*

The following induction proof is based on the shorter proof given in Kröning and Strichman [27]. The main intuition is that whenever a solver backtracks, the search is taken to a fresh region of the search space through the conflict-driven assertion.

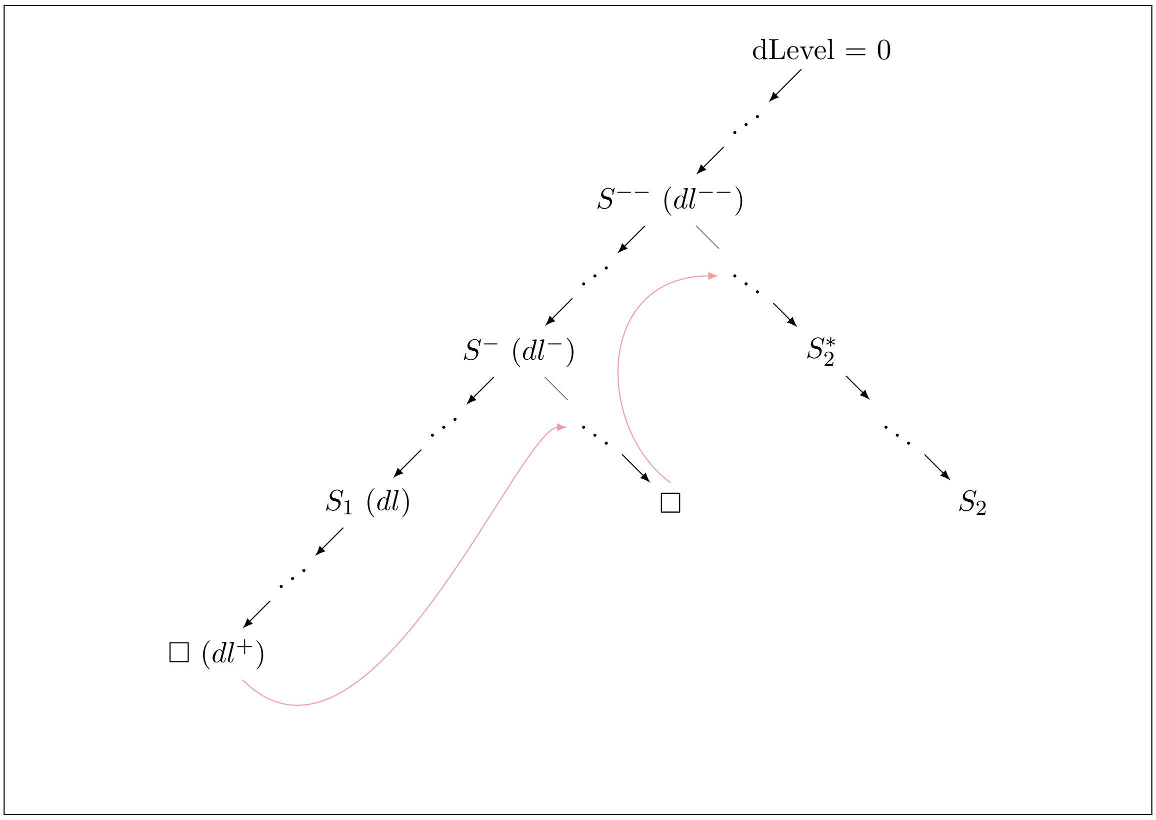


Figure 3.3: An illustration of the induction step of the proof of Theorem 3.2.1.

Proof. Let S_1 be a non-conflicting state at decision level *dl*. Assume furthermore, that there is another, later state S_2 distinct from S_1 where $I(S_1) = I(S_2)$, which would be a necessary requirement for a cycle to occur.

If the solver is to proceed from S_1 to S_2, it must encounter a conflict at a decision level $dl^+ > dl$ which causes the solver to backtrack to a state S^- at a decision level $dl^- \leq dl$.

We claim that, for any such pair of states (S_1, S_2) and any such decision level dl^-, it holds that $I(S_2) \nsubseteq I(S_1)$ and therefore $I(S_1) \neq I(S_2)$. We use induction over dl^- to prove this.

First, take the case where $dl^- = 0$. Let c_{dl^+} be the conflict clause that was learned in the conflict at dl^+. Since, by assumption, c_{dl^+} is an asserting clause, it is unit at decision level 0 and produces a conflict-driven assignment $(a := v) \in I(S^-)$. The conflict-driven assignment must be the opposite phase to an assignment $a := \overline{v}$ that was made on a decision level dl' with $dl < dl' \leq dl^+$. Therefore, it holds that $(a := v) \notin I(S_1)$. Since S^- is at decision level 0, any interpretation $I(S')$ of a subsequent state S' must be an extension of $I(S^-)$ and therefore contain $(a := v)$. Since S_2 is such a subsequent state, $I(S_2)$ must contain $(a := v)$. Since $(a := v) \notin I(S_1)$, it holds that $I(S_2) \nsubseteq I(S_1)$

We can now formulate our induction hypothesis as follows:

For any pair of states (S_1, S_2) where S_1 is earlier than S_2 and the first backtrack after S_1 to a decision level smaller or equal than $\mathrm{dLevel}(S_1)$ *is to a level smaller than dl^-, it holds that $I(S_2) \not\subseteq I(S_1)$.*

It remains to show that $I(S_1) \subseteq I(S_2) \;\Rightarrow\; I(S_2) \not\subseteq I(S_1)$ if the first backtrack to a level smaller or equal than $\mathrm{dLevel}(S_1)$ is to dl^-. Let S^- again be the state after backtracking and assume that $S_2 \geq S^-$. We can distinguish two cases.

- S_2 is reached from the original backtracking state S^- without any further backtracks to decision levels smaller or equal to d^-.

  Then we can reason analogously to the case $dl^- = 0$ that $I(S_2)$ contains a conflict-driven assignment which is not contained in $I(S_1)$. Therefore, it holds that $I(S_2) \not\subseteq I(S_1)$.

- At least one conflict is encountered between S^- and S_2 that causes a backtrack to a decision level smaller than dl^-. Since this case is rather complex, an illustration is given in Figure 3.3. Now let S^{--} be the latest state with $S^- < S^{--} < S_2$, where
  $$\forall S' :\; S^- < S' < S_2 \;\Rightarrow\; \mathrm{dLevel}(S^{--}) \leq \mathrm{dLevel}(S')$$

  Thus, S^{--} is the state right after the last minimum-decision-level backtrack that occurs between S^- and S_2.

  Now assume that $I(S_2) \subseteq I(S_1)$. To reach S_2 from S^{--} it is then first necessary to reach a state S_2^* where $I(S_2^*) \subseteq I(S^-)$ (since $I(S^-) \subseteq I(S_1)$. Therefore, (S^-, S_2^*) is a pair where S^- is earlier than S_2^* and the first backtrack to a decision level smaller than $\mathrm{dLevel}\, S^-$ is to a decision level smaller than dl^-. Furthermore, $I(S_2^*) \subseteq I(S^-)$.

  This contradicts our induction hypothesis, therefore $I(S_1) \not\subseteq I(S_1)$, since we know that such a state S_2^* cannot be found after a backtrack to a decision level smaller than dl^-.

Therefore $I(S_1) \neq I(S_2)$.

□

The introduction of conflict-driven assignments changes the structure of the search process subtly. While in traditional DLL, both values of a variable have to be systematically tried out, in DLL with learning and non-chronological backtracking, we only try out one phase of a variable. If this assignments runs into a conflict, a clause is learned that takes the solver to a new part of the search space automatically after backtracking by virtue of a conflict-driven assignment. All this is conveniently handled by the standard BCP mechanism and needs no special implementation.

3.2.5 Clause-Database Management

A short overview of different clause-database implementations is given in Zhang and Malik [54]. In most cases, a sparse matrix representation is used to represent clauses, i.e., each clause is represented as an array of literals occurring in the clause. Literals themselves are usually represented as integers, an integer i representing the literal v_i and $-i$ representing $\neg v_i$. A common trick that is used in SAT solvers (e.g., MiniSAT) is to use the least significant bit to store the sign, since the variable can then be retrieved simply by a right-shift instead of an "if" operation.

The early SAT solvers Grasp [33] and rel_sat [2] used pointer heavy datastructures to store clauses. This has disadvantages in cache-efficiency since the pointer dereferences lead bad cache efficiency in BCP. Modern solvers usually store the clauses in a large linear array, which necessitates dedicated garbage collection code, but is more efficient overall. Zhang and Malik [54] report some techniques that use zero-suppressed binary decision diagrams [9] or tries [52] to store clauses, but find that the additional overhead is not worth the performance increase. A possible advantage of more structured clause database formats is an easy identification of duplicate and subsumed clauses (clauses whose literals are a subset of another clause's literals) for on-the-fly clause-database simplification.

Besides questions of implementation, systematic issues arise pertaining to the question of how to deal with the growth of the clause database which is exponential in the number of variables in the worst-case. In Grasp [33], a space-bounded diagnosis engine is proposed which makes this worst-case growth polynomial. First an integer k is chosen. Newly learned clauses that have more than k literals are marked for early deletion. They must be kept as long as they define conflict-driven assertions (in order to keep the solver complete), but are deleted immediately afterwards. Since learned clauses encode redundant information encoded in the original CNF instance, deleting clauses does not compromise correctness.

The solver rel_sat [2] proposes to delete clause of low relevance. A clause is considered relevant if few of its variables have changed assignment since the clause was derived. The intuition is to find a way to identify clauses that have a low chance of being used to derive conflicts or unit-assignments in the current part of the search space. BerkMin [19] uses a combined strategy of deleting old clauses with many literals that have not been involved in conflicts recently. The age of a clause is implied by its position on the clause stack. For the clause activity, counters are associated with each clause which count the number of conflicts the clause has helped derive. Counters are periodically divided by a constant, so that more recent activity has relatively higher impact than less recent activity.

3.3 Strategies for Clause Learning

Given a single conflict, there is a number of possible responsible assignments that can be chosen to induce a conflict clause. A very simple strategy would be to choose as a responsible assignment all decisions that have been made so far. Obviously, this assignment is sufficient to produce the conflict. The conflict clause induced by this assignment is on the other hand not very useful. Exactly the same arrangement is unlikely to occur again very often during the search, therefore the clause will be unlikely to be useful in pruning the search space.

There are some properties which seem useful when choosing a conflict clause, some of which are discussed in Zhang et al. [55]. First, the conflict clause should be small since small clauses are considered more powerful in pruning the search space on average. Second, there may be advantages in choosing a responsible assignment as close to the conflict as possible in the implication graph. Finally, the literals in the conflict clause should also have low decision levels, since this increases the gains from potential backjumps.

3.3.1 Unique Implication Points (UIP)

The idea of choosing strong implicants for learned clauses goes back to the seminal paper of Marques-Silva and Sakallah [33], the initial idea being to produce shorter clauses. The idea is illustrated in Figure 3.4. Instead of choosing both $v_2 := 1$ and $v_1 := 0$ for a responsible assignment, we could choose to include $v_0 := 0$ since it is in some sense "stronger" as it implies both of the former assignments. We can formalize this notion by introducing the concept of domination. In an implication graph, a node x at level d is dominated by a node y at the same level iff every path from the decision variable of level d to x goes through y. Therefore, in the above example, $v_3 := 1$ dominates the conflict node, but $v_1 := 0$ does not, since a path $(v_0 := 0, v_2 := 0, v_3 := 1, \square)$ is possible which does not include $v_1 := 0$. Using the concept of domination, we can now give a definition of unique implication points (UIPs).

Definition 15. *A* unique implication point *is a node in the conflict graph that dominates the conflict node.*

In this definition, only nodes that are on the most recent decision level can be UIPs. As an example, consider the implication graph in Figure 3.4 with two UIPs.

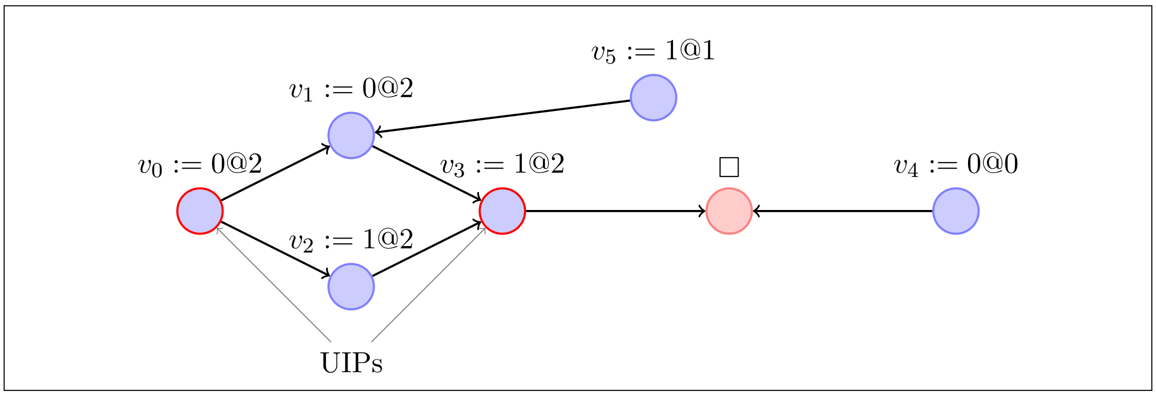


Figure 3.4: A simple implication graph with two unique implication points (UIP).

The idea of choosing a UIP as part of the responsible assignment is important since only learned clauses that contain a UIP variable can be asserting clauses. The most recent decision is always a UIP, which raises the question why we would want to invest the effort into finding other UIPs at all. A possible answer to this question is that UIPs which are closer to the conflict are, in some sense, more general since they encode the more immediate reasons for a conflict. Also, keeping UIPs close to the conflict reduces the number of assignments from earlier decision levels in the responsible assignment hence keeping the overall clause size smaller. Biere [3] makes a very interesting observation on this: Beside the general heuristic gain by smaller clauses, the chances also rise that the second-highest decision level of the clause is earlier compared with a longer clause. This allows for longer backjumps after each conflict which may account for much of the performance gain obtained by using UIPs close to the conflict node.

The concept of UIPs has been extended in Zhang et al. [55] to include nodes on earlier decision levels in a way which depends on the chosen responsible assignment. For the definition, we will borrow the concept of a *separating cut* from Kröning and Strichman [27].

Definition 16. *A* separating cut *in a conflict graph is a subset-minimal set of edges whose removal breaks all paths from the root nodes to the conflict node.*

In Figure 3.4, examples for separating cuts are $\{v_3 := 1, v_4 := 0\}$, $\{v_1 := 0, v_2 := 1, v_4 := 0\}$, or $\{v_0 := 0, v_5 := 1, v_4 := 0\}$.

The assignments that make up a cut always constitute a responsible assignment for the conflict. Given a cut, we can partition the conflict graph into a *conflict side* and a *reason side.* The conflict side contains all nodes that are on a path from a node in the separating cut to the conflict node. The reason side contains all other nodes. We can now generalize the concept of unique implication points to multiple decision levels.

Definition 17. *Given a separating cut T, a node x is a* unique implication point at decision level *d iff any path from a root node either goes through x or through a node y on a decision level $d' > d$, where y is on the reason side of the partition given by T.*

Note that in the case of the most recent decision level, the two definitions are identical, i.e., a UIP is always a UIP at the most recent decision level. We call a UIP u_1 *earlier* than another UIP u_2 on the same decision level if u_2 dominates u_1. The *first UIP* is the earliest UIP on the most recent decision level. The *last UIP* is the first assignment (usually the decision assignment) on the most recent decision level.

The concepts of *last cut* and *first cut* are defined in the following way.

Definition 18. *Given an implication graph with a conflict, the* first cut *is the separating cut that includes exactly the first UIP and those nodes on earlier decision levels that have an edge to the conflict side of the most recent decision level.*

Definition 19. *Given an implication graph with a conflict, the* last cut *is the separating cut that includes exactly the last UIP and those nodes on earlier decision levels that have an edge to the conflict side of the most recent decision level.*

3.3.2 Choosing a Cut for a Learned Clause

At each conflict, it is possible to learn a number of clauses that encode reasons for the conflict. Each separating cut on the implication graph yields a distinct learned clause. In Zhang et al. [55], detailed descriptions and comparisons of different learning strategies are given. The following information about Grasp's and rel_sat's learning strategies will be taken from there, since a more in-depth description is given than in the original papers.

The two first solvers that included learning and backtracking, rel_sat [2] and Grasp [33], have very different clause learning strategies. The solver rel_sat simply adds the conflict clause that is induced by the last cut. The responsible assignment thus includes the decision variable of the most recent decision level and all assignments from earlier levels that have an outgoing edge to a node on the conflict side. Grasp's [33] strategy is described in detail in Zhang et al. [55]. It is unique for two reasons. First, different clause learning strategies are used in the solver depending on the current status of the search where in each multiple clauses are learned for one conflict. Second, some clauses that are learned do not result from conflicts, but constitute general deduction shortcuts. Grasp uses two solver modes that determine how learning is done.

The first, referred to from now on as "decision mode", is active whenever a conflict is encountered as a direct result of a decision. The second, "backtracking mode", is active whenever the solver encounters a conflict when the first assignment on the conflicting decision level was made due to a result of an earlier conflict. Some of this complexity may arise from the fact that Grasp chooses the highest decision level encountered in a conflict clause as a backtracking level instead of the second-highest. Therefore the first assignment made on a decision level may itself be implied by assignments made on earlier decision levels, i.e., the first assignment on a decision level may have antecedents in the implication graph.

Grasp follows the strategy of learning more than one clause from each conflict. In decision mode, the first-cut scheme is used to induce a conflict clause. Additionally, clauses are learned which encode deduction shortcuts for UIPs. In backtracking mode, the same clauses are learned as in decision mode with the addition of a so-called *back-clause*. The back-clause contains only assignments made on earlier decision levels. It can be found by tracing back the implication graph from the conflict node until assignments on earlier decision levels are encountered. This corresponds to the cut where all assignments on the most recent decision level which are on a path to the conflict are on the conflict side and all other assignments on the reason side. After learning the back-clause, the solver backtracks to the highest decision level of any assignment in this cut.

The first-cut scheme that is used in Grasp can also be extended to multiple decision levels using our generalized definition of a UIP. For this, we first determine the partitioning of assignments on the most recent decision level according to the first cut. Then we can step up one decision level and identify a UIP at this earlier level using the partial partitioning on the more recent level below. This can be repeated for an arbitrary number of decision levels. While this seems intuitively useful since it reduces the size of the learned clause, Zhang et al. [55] conclude in an in-depth analysis of learning strategies that the most efficient overall strategy seems to be to just learn one clause corresponding to the first cut, and not try to find UIPs at earlier levels. This strategy also performed better than the Grasp and rel_sat learning schemes. Biere [3] suggests that this increase may result from the increased locality of the search. The first-cut scheme is also the most commonly used learning scheme in newer solvers (e.g, MiniSAT, RSAT).

3.4 Efficient Datastructures for BCP

In DLL algorithms, *Boolean Constraint Propagation* (BCP),i.e., the exhaustive application of the unit rule, is usually by far the most time consuming subtask. It is estimated that BCP takes up about 90% of the runtime in a typical solver [34]. Therefore, improving the efficiency of this step is one of the most crucial aspects in engineering a fast implementation.

One of the main determining factors for the efficiency of any implementation is the choice of datastructures. In BCP, the areas of special interest are those which are involved in determining the state of a clause at any point in the search process, that is, to determine when a clause becomes unit, conflicting, or satisfied under the current partial assignment. Newer SAT solvers do usually not distinguish between clauses that are satisfied and unresolved clauses. The additional overhead of managing such a clause list is not worth the benefits.

We can broadly distinguish between busy and lazy datastructures for determining the state of a clause. Busy datastructures update the state of a clause immediately after one of its literals has been assigned. Early SAT solvers such as GRASP [33] used this approach. A major performance improvement was obtained by using lazy datastructures. They perform more work at each state update, but such updates are not performed every time a clause literal is assigned. Compressing more related work into steps that happen less often increases space locality. Clause literals are usually stored in contiguous blocks of memory. Accessing more than one literal at each state update therefore accesses the cache instead of the main memory, which leads to an overall performance improvement.

The average numbers of accesses per variable assignment that are presented in the following discussion are taken from [53].

3.4.1 Counter-Based Approaches

Most prominent in the category of busy datastructures is the counter-based approach, in which literal assignments are represented by counters associated with each clause. In Zhang and Malik [53], the first use of this approach is traced back to Crawford and Auton [11]. Clause counters are updated when a variable is assigned, and have to be undone during backtracking.

Grasp [33] uses a simple version of the counter-based approach, counting the number of satisfied (c_s) and unsatisfied (c_u) literals for each clause, and checking these values against the total number of literals n_l in the clause. Unit clauses and conflict clauses can then be efficiently determined.

If $c_s = 0$ and $c_u = n_l - 1$, then the clause is unit, if $c_u = n_l$, it is a conflict clause. For a random SAT instance with n variables, m clauses, and an average of l literals per clause, this necessitates an average of $\frac{l*m}{n}$ counter updates per variable assignment.

Another variant of this scheme is proposed in Zhang and Malik [53] which is slightly more efficient. In this version, each clause is associated with only one counter that corresponds to the number of literals that are not unsatisfied. Then, the average number of accesses is decreased to $\frac{l*m}{2n}$. Additionally, since less space is needed to represent the counters, this approach is slightly more cache efficient.

3.4.2 Head/Tail Lists and Literal Watching

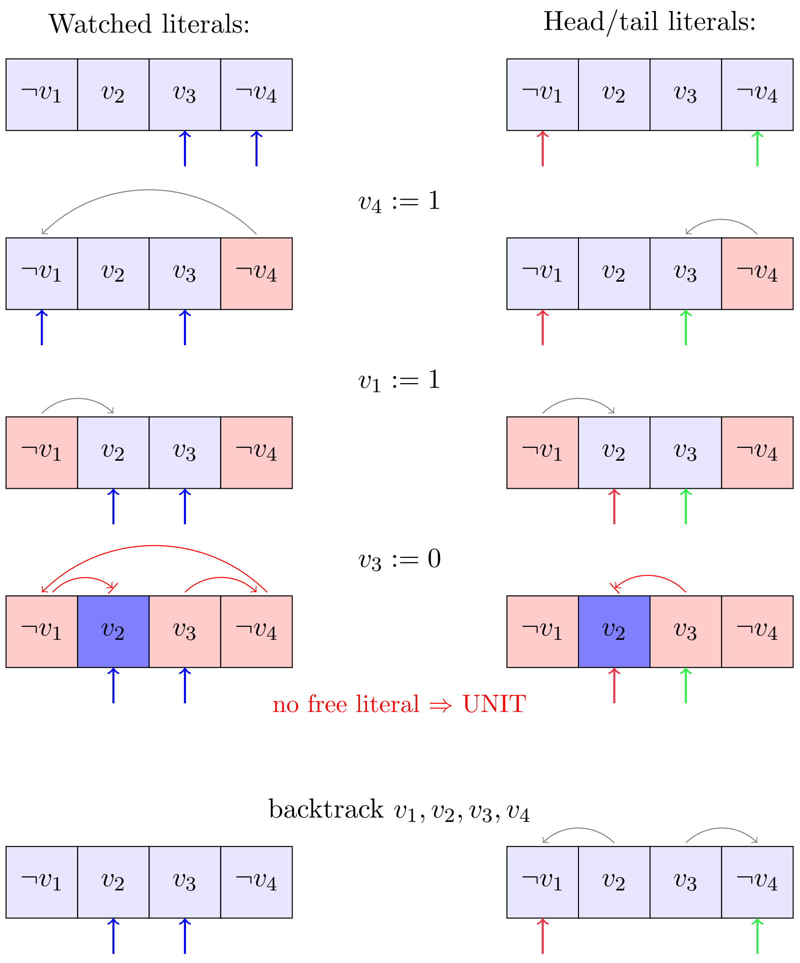
Lazy datastructures use the following basic idea to effectively decide the state of a clause. As long as there are at least two distinct literals that do not evaluate to false in a clause, then this clause is either satisfied or unresolved. This can be monitored by keeping two references to distinct satisfied or unassigned literals, the so-called *watched literals*. When one of them gets assigned to a value that dissatisfies the literal, the reference is moved to another satisfied or unassigned literal which becomes the new watched literal. If no such literal can be found, then the clause is either unit or conflicting, depending on whether the other watched literal points to an unassigned or unsatisfied literal.

Literal references can be implemented by managing occurrence lists. An occurrence list of a literal stores references to each clause in which that literal is watched. If an assignment is performed, the occurrence lists of the newly dissatisfied literal is visited. Moving a watched literal inside a clause can be done by removing a clause from its old occurrence list and appending it to the list of the newly watched literal.

The idea of referencing two literals to determine the status of a clause was first presented in the form of *head/tail (H/T) lists* in the SAT solver SATO [52]. As the name implies, two lists with references to literals are kept here, the head lists and the tail lists. The head list is initialized with pointers to the first literal of each clause, the tail list with the last literal. When a literal that is referenced in one of those two lists is assigned a value, the reference is removed from the list, and a new unassigned literal is searched for. In the head list, this search is performed in the direction of the last literal, in the tail list, the direction is reversed. If a satisfied literal is encountered, the clause is declared satisfied. If a new unassigned literal is encountered, it is added as a new reference to the appropriate list, if none can be found, the status of the clause is updated according to the value of the clause's second watched literal. If the other watched literal is satisfied, the clause is declared satisfied, if it is unsatisfied, the clause is conflicting, and if the other literal is unassigned, the clause has become unit and the other watched literal is the new implied assignment.

Backtracking for the H/T datastructure requires moving literal references back against their normal directions in the head and tail lists as far as possible. Caching can be used to associate backtracking levels with certain pointer configurations so that backtracking of a clause's state takes constant time, but this also increases memory consumption.

Another scheme was proposed in the SAT solver Chaff, the *watched-literal* (WL) scheme [34]. As in H/T, two references to literals are kept for each clause. The difference



Consider the example clause

$$\neg v_1 \vee v_2 \vee v_3 \vee \neg v_4$$

above and its state changes during BCP. In the watched-literal (WL) scheme (left), the watch pointer movement direction can be arbitrary. In the head/tail-list (H/T) scheme, the head literal always moves left-to-right, and the tail literal right-to-left.

During backtracking, no work needs to be done for WL. For H/T, the head and tail pointer have to be moved to the leftmost and rightmost non-conflicting literal respectively.

Figure 3.5: Chaff's watched literals and SATO's head/tail lists.

between the two schemes is that in WL, no special order or search direction is imposed on the literals. The only requirement when searching for a new position is that the new literal is unassigned or satisfied.

This may lead to more time being spent in search for a new unassigned literal whenever a clause's status is updated, but this is a operation that is highly local in memory and therefore very cache efficient. Another advantage of WL is that it removes the need for changing any clause's references during backtracking.

To summarize, the watched-literal scheme as implemented in Chaff [34] performs the following steps when processing an assignment to an opposite-phase watched literal:

Step 1 Search for a new unassigned or satisfied literal.

Step 2 If such a literal exists and it is not the second watched literal of the clause, move the watched literal to the new position by removing the clause reference from the old watch list and appending it to the watch list of the new watched literal.

Step 3 If no such literal exists, check the status of the other watched literal:

3.1 If the other watched literal is unassigned, the clause is unit.

3.2 If the other watched literal is unsatisfied, the clause is conflicting.

In a SAT solver that uses the WL scheme, performing the above steps usually takes up most of the runtime. It is therefore worthwhile to investigate highly efficient implementations as is done in Biere [5]. A popular approach used in many solvers is to associate each literal with a list (or stack) of so-called *watcher* datastructures. These watchers keep a reference to a clause and its watched literals. Biere [5] proposes instead to reference the clause as a whole and keep the watched literals at the first and second position of the clause respectively. This avoids the overhead of separate watcher datastructures and yields an improved cache efficiency. Figure 3.6 illustrates this technique.

3.4.3 Special Handling of Small Clauses

In many practical problem instances, binary and ternary clauses, i.e., clauses with two or three literals take up a considerable percentage of the input instance. Given a random variable assignment, the chance of such a clause becoming unit is relatively high. Unit-rule applications over binary clauses therefore may make up a considerable percentage of BCP.

For binary clauses, determining their status with the standard WL technique entails considerable overhead compared with a more direct implementation. One possibility would be to add vectors of direct implications, which—similar to the implication graph in 2-CNF—store for every literal l all other literals l' where $l \vee l'$ is a binary clause in the clause database. Whenever a variable is assigned a value, all resulting assignments due to binary clauses can then be processed immediately. If one of the resulting assignments is impossible since the opposite value is already assigned to the destination variable, a conflict can be concluded.

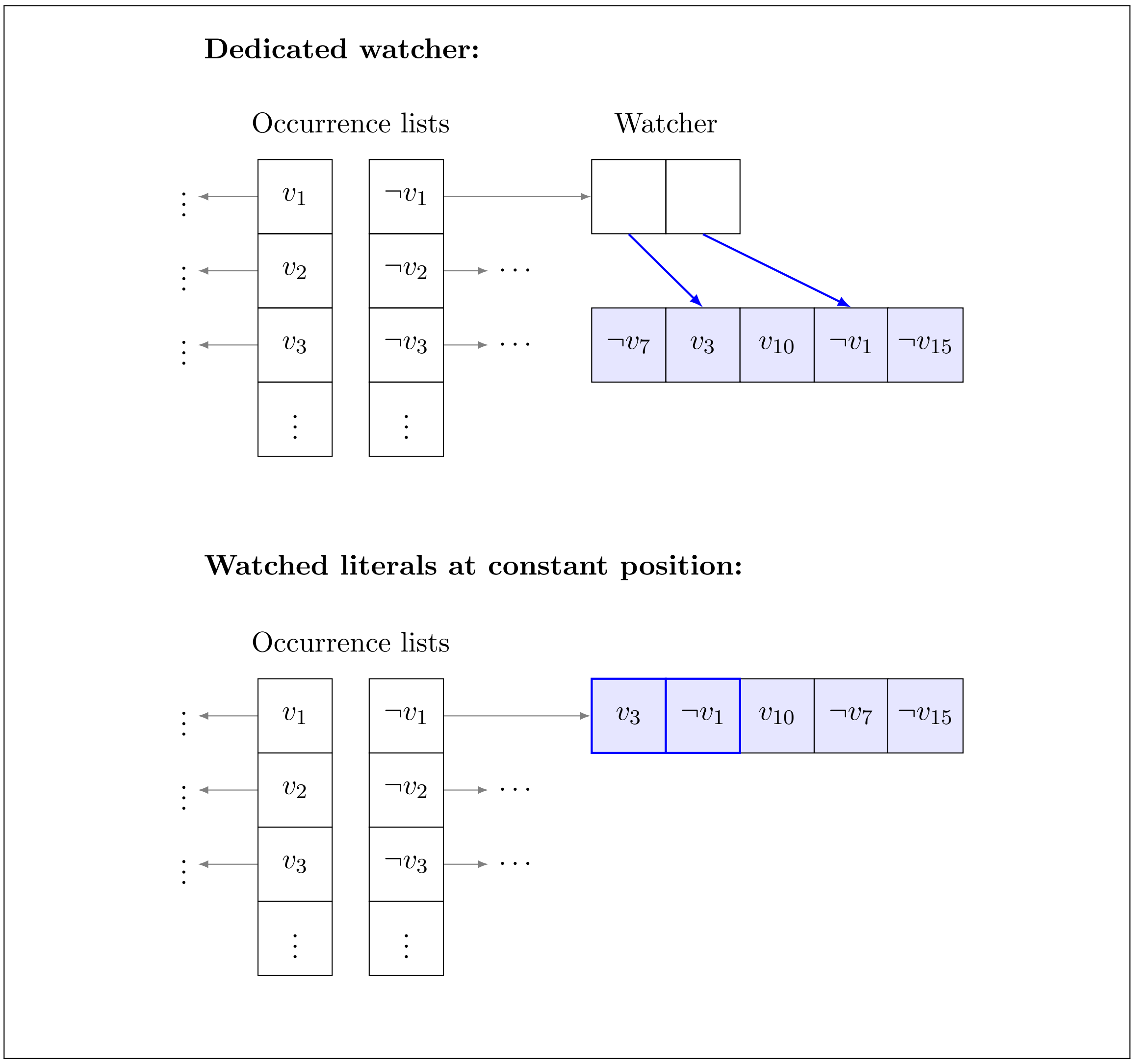


Figure 3.6: Different implementations of the watched-literal scheme.

In the SAT solver FUNEX, clauses are divided into three classes (binary, small, and large), and only large clauses use the watched literal scheme (for a detailed discussion see Biere [3]). MiniSat [39] uses mixed watch lists for literals, that contain both references to clauses and directly implicated literals from binary clauses. The reason given for this implementation choice is that, if stored separately, all direct implications are performed together either before or after the watched-literal BCP step, which is reported to lead to slightly less useful conflict clauses. On the other hand, the SAT solvers FUNEX and NanoSAT make use of implication lists exactly for this reason, i.e., to be able to process binary clause implications before the actual BCP step. The reason given here is that the binary BCP step is less expensive and should therefore be used exhaustively before going on to the more expensive watched-literal BCP step.

3.5 Variable Selection Heuristics

The DLL procedure is highly sensitive to the choice of decision variables. For satisfiable instances, an optimal decision heuristic would be able to produce a result in a linear number of DLL steps by incrementally choosing decision variables from a satisfying assignment. Of course, since SAT is NP complete, the problem to find any such perfect heuristic for an input formula must be NP complete itself, thereby moving the complexity of SAT into the decision heuristic. While such a perfect heuristic is not practically interesting, it is still possible to find heuristics that work well in the average case for certain classes of problems.

For unsatisfiable instances, a good decision heuristic would choose variables that are expected to lead to conflicts as early as possible. This keeps the search tree from expanding unnecessarily by closing down unfruitful parts of the search space early in the search process. Additionally, causing conflicts early produces short conflict clauses. Short conflict clauses are more powerful than long conflict clauses since they prune larger parts of the search space. In many SAT implementations, long learned clauses are removed from the clause database after a while or not even stored for exactly this reason.

Another important consideration when choosing a decision heuristic is its efficiency. Often a trade-off has to be considered between the predictive power of a heuristic and the runtime that is consumed by its computation.

3.5.1 Early Heuristics

Early decision heuristics were usually greedy in the sense that they tried to maximize the number of implications that a given variable assignment would produce. In order to estimate the consequences of a variable assignment, some heuristic function over the input CNF-formula and the current partial assignment is used. Examples are *Böhm's heuristic* and the *Maximum Occurrences on clauses of Minimum size heuristic* (MOM), both of which are briefly described in Marques-Silva [32]. Both of these choose literals which satisfy the highest number of possible clauses considering only unsatisfied clauses of minimum length. Moreover, both try to choose a balanced variable, for which both literals fulfill this criterion to some extent. They differ in the exact balancing method that is used. Böhm's heuristic just calculates a linear weighting of both literals' scores for a variable, whereas MOM uses a non-linear function.

Dynamic largest independent sum (DLIS), originally presented in Marques-Silva [32], is a conceptually simpler, but nevertheless effective heuristic based on counting literals. Here, the decision assignment is chosen that makes the literal with the highest independent occurrence count among unsatisfied clauses evaluate to true. The term "independent", here, refers to the fact that the score of the opposite-phase literal does not influence the decision, i.e., for choosing a phase, DLIS is a greedy heuristic, not a balanced one.

3.5.2 Second-Order Heuristics

While earlier decision heuristics were functions of the current state of the solver, the SAT solver Chaff [34] introduced second-order heuristics that are functions not only of the current, but also of the preceding states of the solver.

Specifically, a second-order heuristic called *Variable State Independent Decaying Sum* (VSIDS) was introduced, which depends closely on the conflict analysis step that Chaff uses for clause learning and non-chronological backtracking.

VSIDS essentially estimates a literal's likelihood to produce conflicts by pushing a sliding window over all literals involved in recent conflicts. Each literal on the variables in the input formula is associated with two counters that are initialized to zero. Whenever a conflict is encountered, the counters for the literals occurring in the conflict clause are increased. In the decision step, a branching literal with maximum score is chosen for assignment. Periodically, all counters are divided by a constant. This leads to older conflicts becoming less relevant for a literal's score, which, in turn, increases the locality of the search.

In the SAT solver BerkMin [19], a number of changes are made to the VSIDS procedure. The most important change is that the choice of a decision literal, while still in general guided by the maximum score, is confined to the so-called *top clause*. The top clause is the most recently learned clause that is not satisfied.

The idea behind this procedure is to increase the mobility of the search, that is, to increase the speed at which the search refocuses on a new part of the search space. If the search enters a new region, e.g., by backtracking non-chronologically, VSIDS may make bad decisions for a number of iterations due to the fact that its literal scores change only slowly to reflect the circumstances of the new part of the search space. BerkMin's strategy of choosing a literal of the top-clause as decision variable reduces this lag considerably by further promoting variables involved in recent conflicts. A naive implementation can lead to a large percentage of the runtime being spent searching for the top clause. Caching of the location of the top clause in the clause stack eliminates this problem.

Another change to VSIDS is that, in BerkMin, instead of just incrementing the scores of the conflict clause's literals, the solver steps backwards through the implication graph and increases literal scores of clauses that were involved in producing the conflict. If a certain literal occurs multiple times in the clauses, its score is also increased more than once. This entails a slightly higher overhead for the decision heuristic, but allows for more finely-grained estimation of a literal's tendency to lead to conflicts.

3.6 Restarting and Randomizing the Search

As has been mentioned before, DLL-based SAT procedures are highly sensitive to the choice of decision variables. Good variable orderings may yield exponential speed-ups in overall solving time compared to bad ones, but the computation of such orderings is a hard problem in itself. Usually, some kind of heuristic is used to prevent the solver from entering fruitless regions of the search space, but these are far from infallible. The problem is mitigated somewhat by non-chronological backtracking and learning, which prune large parts of the search space, but the problem still remains. Biere [5] notes that this is especially a problem in industrial instances, since they are mostly either easy to refute or easy to prove, given the right variable ordering.

In Gomes et al. [20], this behaviour is referred to as a "heavy-tailed cost distribution" since there is a non-negligible chance that a variable ordering will lead to very long run-times. They argue for introducing *random restarts* into combinatorial search algorithms in order to increase overall robustness by increasing the chances that a "good" search path will be chosen. In this technique, some transient randomness is introduced into the search process and the search procedure is reset at some points. Random restarts were implemented in Chaff [34] and are a feature of most competitive DLL SAT solvers since then. They form an especially effective combination with SAT solvers since information can be retained in between runs through learned clauses which implicitly encode parts of the visited search space in the clause database. In order to retain completeness, a solver must perform successively longer runs since, in the worst-case, the search space will have to be fully explored.

Introducing randomness into the decision heuristic can prevent the DLL procedure from being trapped in fruitless regions of the search space, and it allows for a broader exploration of the same. This, in turn, allows decision heuristics to work more effectively. A similar argument was presented as to why "throwing away" parts of the search space in backtracking seems to improve solving efficiency.

In Goldberg and Novikov [19], the authors of the solver BerkMin raise the issue of clause-database symmetrization. They argue that restarts introduce asymmetry into the clause database since many variables may not be tried out in both phases, but just in one before a restart is triggered. If the decision heuristic does not account for restarts, it is possible for the solver to enter the same phase for these decision variables again. This leads to a situation where some variables mostly occur in one phase in learned clauses. For phase selection, Goldberg and Novikov [19] propose a modified strategy, where the decision *variable* is chosen according to the VSIDS scheme among the top-clause literals, but not its *phase*. Counters are associated with each literal which count its occurrence in conflict clauses, and the phase is chosen that occurs more frequently in the clause database. If this assignment leads to a conflict the opposite phase literal may be learned in a clause thereby symmetrizing the database.

Another important question when using restarts is the restart schedule that is used. Usually, a restart is triggered after a fixed number of conflicts. This limit increases with every restart in order to guarantee the SAT solver's completeness. Some work has been done on trying to determine good restart policies. Earlier restart-limit sequences were usually strictly monotonically rising. MiniSAT [39], for example, uses a geometric progression that starts with a clause limit of 100 and increases by a factor of 1.5 after each restart.

More recent solvers such as RSAT 2.0 [36] or PicoSAT [5] use restart-limit sequences that do not continually grow, but return to smaller restart-limits at times. In RSAT, the restarting limit sequence is based on the Luby sequence, which gives an optimal speed-up for Las Vegas algorithms [31]. PicoSAT uses a similar scheme which uses two limits, an inner limit and an outer limit. The solver triggers restarts upon reaching the inner limit which is increased geometrically. When the inner limit reaches the outer limit, the inner limit is reset to an initial value and the outer limit is increased. As can be seen in Figure 3.7, PicoSAT's strategy concentrates on very short restart intervals that only rise slowly globally, while RSAT's strategy intersperses very long runs with very short ones.

Another interesting technique that works in combination with restarts is phase saving [35]. Phase saving means that upon backtracking or restarts, the phase of each assigned variable that is undone is saved in an array. The next time a decision is made on of these variables, the phase indicated by the array is used first. If no array entry exists for a variable, the default phase-selection heuristic is chosen. The motivation is that the solver may discard regions of the search space in non-chronological backtracking and during restart which already encode satisfiable assignments for subproblems.

After backtracking or restarting, when the solver decides upon variables of the discarded search space, it automatically reenters parts of the search space similar to the one that was discarded. Biere [5] argues that phase saving "turbo charges" restarts. The solver immediately reenters regions of the search space that it left due to the restart and thus continues the search from where it left off.

3.7 Simplifying CNF formulas

Smaller CNF input formulas tend to produce shorter run-times for most SAT solvers. This relationship is, of course, not true in general. The idea of clause learning, for example, is based on enriching the input formula with redundant clauses to speed up the solving process. There are very small problems that are hard for SAT solvers as well as large problems that are very easy. Still, it is useful to concentrate on the size of a formula since it is easy to determine and gives some estimation of SAT solver running time. Simplification can be used to replace the original formula with an equisatisfiable formula that is smaller in the number of clauses or variables in order to keep the runtime of a SAT solver heuristically low.

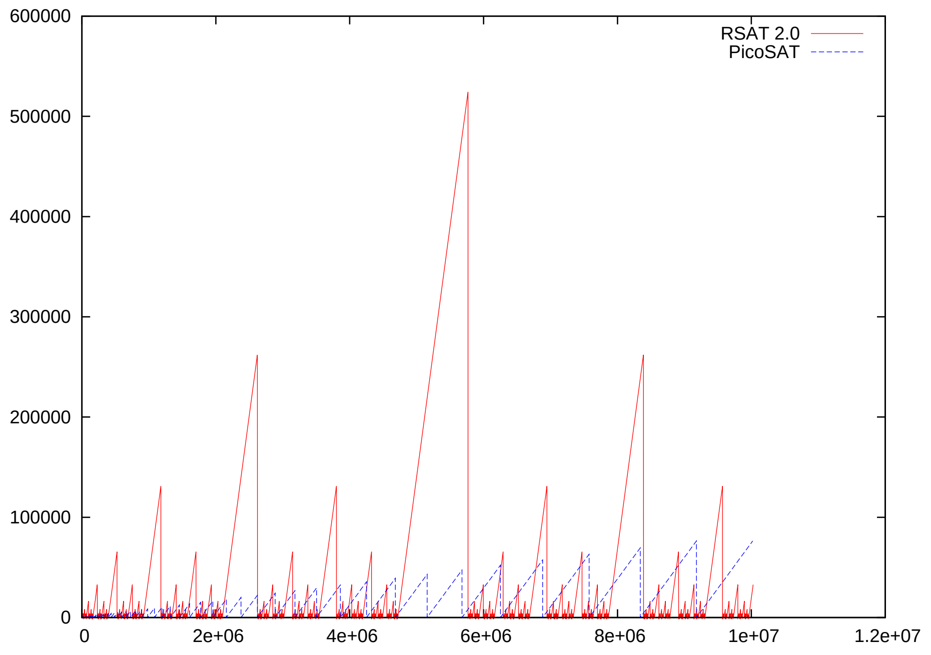


Figure 3.7: Comparing PicoSAT's and RSAT's restarting schemes. The x-axis shows the number of conflicts, the y-axis shows the restart-limit.

3.7.1 Preprocessing SAT Instances

Many efficient preprocessing steps may be performed before the problem is actually translated into a SAT instance given in CNF. These may include domain-specific simplification techniques, efficient translation techniques into propositional form, and reduction techniques that work on structured formulas, such as the identification of equivalent sub-formulas. Other techniques directly work on the resulting CNF and may be used in addition with other prior reduction techniques. We will concentrate only on the latter in this section.

Some CNF preprocessing steps trace back to deduction rules in the original DP [12] procedure. One is the application of the pure literal rule, i.e., the removal of all clauses that contain a literal occurring only in one phase in the CNF, which can be applied as a light-weight preprocessing step. The other is the application of the *elimination rule for atomic formulas*, reintroduced as a preprocessing step in the ZBDD-based SAT solver ZRes [9], the quantified Boolean formula procedure (QBF) Quantor [6], and the dedicated CNF preprocessor Niver [42]. Here, a variable is chosen, and the clauses containing that variable are replaced by their resolvents in the original instance. In general, this step does not reduce size of the formula, it may, on the contrary, increase it exponentially. But if care is taken in the selection of variables resolved upon, the size of the formula can be decreased and variables removed in the same step. Used as a preprocessing step, the elimination rule is referred to as *clause distribution.*

For both of these techniques, the integration into the main solving step has proved to be too expensive in modern implementations, but as preprocessing steps, they are efficient and light-weight ways of eliminating variables and clauses from formulas. Preprocessing is always a trade-off between the time spent in preprocessing and the expected benefits of running a SAT solver on the simplified formula. In Eén and Biere [15], three less trivial preprocessing techniques and efficient implementations are described, all of which are included in the dedicated SAT preprocessor SatELite: Removal of subsumed clauses, strengthening clauses by self-subsuming resolution, and variable eliminations by substitution.

The authors note that clause distribution produces many subsumed clauses. Let $\mathrm{lit}(c)$ be the set of literals of a clause c, then a clause c_1 *subsumes* another clause c_2 if $\mathrm{lit}(c_1) \subset \mathrm{lit}(c_2)$. Of course, subsumed clauses are not only an artifact of clause distribution, but can occur naturally as part of the encoding of a problem. Since c_1 will be in conflict whenever c_2 is in conflict, and c_1 will be unit whenever c_2 is unit otherwise, the addition of c_2 in a SAT instance containing c_1 does neither change the satisfiability of the the original formula nor add useful deduction shortcuts. The only significant way in which c_2 could influence the solving process is by influencing the decision heuristics in some way. Clauses that are subsumed can thus be removed from the instance, producing an equivalent instance with less clauses that is easier to solve.

In *self-subsuming resolution*, pairs of clauses are identified where one is almost subsumed by the other, except for a literal that occurs in opposite phases in both clauses.

Definition 20. *Let c and c' be clauses and x a propositional variable. Remember, that*

$\otimes$ *is the binary resolution operator for clauses. The clause c is* self-subsumed by c' w. r. t. x *iff* $\mathrm{lit}(c \otimes_x c') \subset \mathrm{lit}(c)$.

Clauses that are self-subsumed by another clause w. r. t. to a variable x, can be replaced in the instance with their resolvents over x. As an example, consider $c_1 = \neg a \vee b \vee x$ and $c_2 = \neg a \vee \neg x$. The resolvent of both clauses would be $c_r = c_1 \otimes_x c_2 = \neg a \vee b$, which subsumes c_1. Therefore, c_1 can be replaced by c_r in the SAT instance, effectively removing a literal from the original clause.

The last technique presented in Eén and Biere [15], the *variable eliminations by substitution* rule is motivated by the fact that many SAT instances are encoded via the structure-preserving Tseitin transformation [47], which will be discussed in detail in Chapter 4. The Tseitin transformation is used to transform arbitrary propositional formulas or circuits into equisatisfiable CNF-formulas by introducing new variables for subformulas or gates. As an example, consider the simple formula

$$(x \vee \neg y \vee (u \wedge \neg v)).$$

A possible application of the Tseitin transformation would introduce a new variable z for the subformula $(u \wedge \neg v)$ and produce the equisatisfiable formula

$$(x \vee \neg y \vee z) \wedge (z \Leftrightarrow (u \wedge \neg v))$$

which is subsequently transformed into the CNF

$$(x \vee \neg y \vee z) \wedge (\neg z \vee u) \wedge (\neg z \vee \neg v) \wedge (z \vee \neg u \vee v)$$

The newly introduced variables are functionally fully dependent on the variables occurring in the subformulas they name. If a naming variable such as x is removed in a clause distribution step, the effects of this renaming process are partially undone. This introduction and subsequent removal of variables introduces some redundant clauses. The variable elimination by substitution rule directly tries to reconstruct gate definitions used by the Tseitin transformation. If the preprocessor decides to remove a naming variable v of such a definition, instead of using clause distribution, the Tseitin transformation is locally undone by replacing all clauses containing v with the resolvents between the gate definitions and all other clauses where the naming variable occurs. Note that this is just a subset of all possible resolutions on that variable. Eén and Biere [15] prove that restricting the resolutions in this way yields a formula that is equivalent to the result of performing clause distribution on v and thus equisatisfiable to the original instance.

In the above example, the clauses that define the subformula $\{(\neg z \vee u), (\neg z \vee \neg v), (z \vee \neg u \vee v)\}$ would be resolved with the remaining clause $\{(x \vee \neg y \vee z)\}$ in order to produce the formula

$$(x \vee \neg y \vee u) \wedge (x \vee \neg y \vee \neg v),$$

which would have been the result of directly translating the original formula into CNF.

Directly using clause distribution, would have produced the CNF

$$(x \vee \neg y \vee u) \wedge (x \vee \neg y \vee \neg v) \wedge (u \vee \neg u \vee \neg v) \wedge (v \vee \neg u \vee \neg v).$$

In this case, it is easy to just check the newly produced distributed clauses for validity and thus remove the redundant clauses $(u \vee \neg u \vee \neg v)$ and $(v \vee \neg u \vee \neg v)$. In more complex examples, clause distribution can produce clauses which are not valid, but are still redundant.

Note that the combination of the Tseitin transformation and subsequent variable elimination by substitution is functionally similar to approaches such as Boy de la Tour [8], where a translation procedure is presented that only introduces naming variables if the introduction reduces the overall number of clauses. The variable elimination by substitution rule is more general in the sense that it can work on any given instance in CNF, no matter how it was translated.

Preprocessing as a means of speeding up the search process has proven to be a highly effective technique. The combination of the dedicated preprocessor SatELite with an efficient SAT solver has won the SAT 2005 competition on industrial benchmarks [43], and preprocessing has since been integrated into some of the fastest state-of-the-art solvers, such as new versions of MiniSAT [16] and PicoSAT [5].

3.7.2 On-the-fly Clause-Database Simplifications

The techniques presented here are not only useful before the search process is started, but can be applied during the search. Most prominently, if a solver uses a restarting scheme, a simplification step can be performed at each restart. This has been implemented with good results in MiniSat 2.0 [16]. Moreover, earlier versions of the same solver already introduced another form of on-the-fly simplification: conflict-clause minimization.

In conflict-clause minimization, the self-subsuming resolution rule is invoked whenever a conflict clause c is learned. Recall that each of the literals l in c corresponds to an opposite-phase assignment of the same variable in the conflicting implication graph. Each of these assignments, in turn, is associated with a clause c' containing $\bar{l}$ that induced the assignment via the unit rule. Since c and c' contain an opposite-phase literal, c is a candidate for being self-subsumed under c'.

Self-subsumption can be checked easily with the algorithm shown in Figure 3.3. In essence, each literal in a newly learned clause is checked for possible removal under the self-subsumption rule. Intuitively, a clause that encodes the reasons for an assignment does not need to encode that assignment itself.

Algorithm 3.3: MiniSAT's on-the-fly clause minimization.

```
input  : A clause $c$
output: A possibly reduced clause $c'$

for each literal $p \in c$ do
    if lits(reason($\overline{p}$)) $\setminus \{\overline{p}\} \subseteq$ lits($c$) then
        mark($p$);
$c' \leftarrow$ removeMarked($c$);
return $c'$;
```

Another application area for clause-database simplification is in incremental SAT problems. In incremental SAT, a series of closely-related SAT instances are solved one after another. An example of this are BMC runs with an increasing depth-bound k. Before each run, a formula is created that represents the formula of the last run, expanded by a new instance of the sequential circuit. A short study of applying simplifications between runs of an incremental SAT solver can be found in Eén and Biere [15].

Chapter 4

Solving SAT in Circuits

Many applications of SAT solving feature problems formulated as constrained Boolean circuits, i.e., circuits with some signals set to a fixed value. In order to solve these problems with traditional SAT solvers, they must first be translated into propositional form, and then flattened into CNF, which yields an exponential increase in formula size in the worst case. This problem of exponential growth can be sidestepped by solving the satisfiability instance not on the original propositional formula, but on a satisfiability-equivalent formula with additional variables introduced for gates. Although some structural information is still encoded in the resulting CNF, CNF-based solvers usually do not make use of this information.

This is a problem since the structural information could be used to speed up the solving process. One possible approach is to use circuit information in order to change the circuit-to-CNF translation, either by producing a smaller translation or by encoding additional knowledge about the circuit structure into the CNF. Another possibility is to forego the use of a purely CNF-based solver altogether and explicitly work with structural information in the solving process. Intuitively, we expect that the added structural information should lead to better heuristics, simplification techniques, and search strategies. On the other hand, the structurally simple format of CNF allows for highly efficient implementation techniques and low-level optimizations, and given the more complex format of a Boolean circuit, it may be harder to achieve the same level of efficiency.

This chapter aims to describe methods which try to speed up CNF-based solvers by using sophisticated translation techniques as well as methods which directly use circuit information in the solving process. Section 4.1 describes approaches which aim to improve the efficiency of circuit-to-CNF translations for use with traditional CNF-based SAT solvers. Section 4.2 deals with hybrid approaches, which combine CNF and circuit representations, typically by enhancing a traditional SAT-Solver with circuit-specific structural information, and the last section, Section 4.3, presents solvers that exclusively work on a circuit and do not include CNF-based components.

4.1 Efficient Circuit-to-CNF Translations

One possibility of performing satisfiability checks on Boolean circuits is to transform them into conjunctive normal form (CNF) and applying a CNF-based SAT solver. This procedure is especially interesting since most current SAT solvers are CNF-based, and their performance has improved drastically over the last few years [44]. Furthermore, CNF-based solvers are an active research topic, and since they can usually be used as black-box components in larger systems, such systems can easily benefit from possible future improvements.

In the 2007 SAT Competition [44] and the SAT Race 2007 [45], solvers that work on a simple circuit format were evaluated. In both instances, solvers that preprocess and translate the circuit into CNF outperform dedicated circuit solvers. While this may be an indication that CNF-based solvers coupled with efficient preprocessing and translation techniques may be a more efficient choice for industrial instances in general, it is very likely that some of the success can be attributed to the high amount of low-level tuning that is employed in state-of-the-art CNF-based SAT solvers.

It is interesting to note, that the preprocessing techniques pioneered in the tool SatELite [15] and later integrated into MiniSAT [16], one of the most efficient CNF-based solvers, tries to reconstruct some of the circuit structure. This can be considered as an indicator of the importance of finding efficient translations from circuit instances to CNF.

4.1.1 The Tseitin Transformation

The translation from a constrained Boolean circuit into a propositional formula is straightforward. In Chapter 2, a very simple procedure was already presented. The constrained circuit shown in Figure 4.1, for example, could be directly translated into the formula

$$\phi = (c_1 \wedge c_2 \wedge c_3) \vee (c_4 \wedge c_5 \wedge c_6) \vee (c_7 \wedge c_8 \wedge c_9)$$

The procedure presented earlier would actually translate the circuit into the formula $o \Leftrightarrow \phi$, but since we only have one output, and since this only output is constrained to true, we can choose the above formula as a translation.

When performing a straightforward translation from a circuit structure's propositional logic formula into CNF, the CNF can, in the worst case, grow exponentially in size compared to the original formula. The above formula, as an example, is in disjunctive normal form (DNF), i.e., it is a disjunction of conjunctions. Naively transforming such a formula into CNF necessitates the repeated application of the law of distributivity, which yields an exponential increase in formula size.

The conventional way of dealing with this problem is by using *renamings.* Intuitively, the process of renaming replaces some subformulas of the original formula with newly introduced literals, and conjuncts the result with a definition for each of the introduced symbols. A formula $\psi = (a \vee (a \wedge (b \Rightarrow c)))$ may thus become $\psi' = (a \vee g_1) \wedge (g_1 \Leftrightarrow (a \wedge (b \Rightarrow c)))$ if we chose to rename the subformula $\phi_1 = (a \wedge (b \Rightarrow c))$. Additionally, we

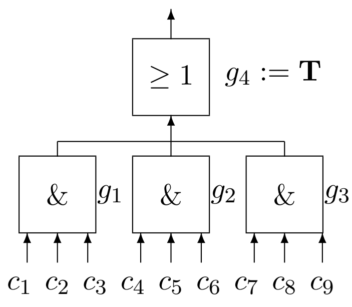


Figure 4.1: A small example circuit.

could include the formula $\phi_2 = (b \Rightarrow c)$ in our set of renamed subformulas, which would yield the result $\psi'' = (a \vee g_1) \wedge (g_1 \Leftrightarrow (a \wedge g_2)) \wedge (g_2 \Leftrightarrow (b \Rightarrow c))$.

It must be noted that, as the signature of the formulas changes in the renaming process, logical equivalence between the original formula and the renamed result is lost. Still, satisfiability equivalence is maintained, i.e., the original formula is satisfiable iff the renamed formula is satisfiable. The same procedure can be easily extended to constrained Boolean circuits. For a constrained Boolean circuit (C, τ) with $C = (\mathcal{G}, \mathcal{E})$ being a Boolean circuit, we can build a propositional formula

$$\Phi_C := \Phi_{\text{constr}} \wedge \Phi_{\text{def}}$$

where the constraints are encoded as

$$\Phi_{\text{const}} := \bigwedge \{\, v \mid (v := \mathbf{T}) \in \tau \,\} \quad \wedge \quad \bigwedge \{\, \neg v \mid (v := \mathbf{F}) \in \tau \,\},$$

and the gate definitions are encoded as

$$\Phi_{\text{def}} := \bigwedge \{\, v \Leftrightarrow f_v(c_1, \ldots c_n) \mid v := f_v(c_1, \ldots c_n) \in \mathcal{E} \}.$$

We call the variables on the left-hand side of the equivalences in Φ_{def} *naming variables* and the equivalences themselves *definitions*. In Φ_{def} above, definitions are given for every single gate in the circuit. Alternatively, we can choose to collapse an arrangement of gates into one single definition, e.g., for the example presented in Figure 4.1, we could combine gates g_4 and g_1 into a single definition by introducing an equivalence

$$g_4 \Leftrightarrow ((c_1 \wedge c_2 \wedge c_3) \vee g_2 \vee g_3).$$

It is then easy to see that there are a number of possible renamings for each formula. With such different renamings, the size of the resulting CNF formula as well as the length of the shortest proof in a given calculus can change drastically. Producing small formulas may *not* lead to smaller run-times, but it is still interesting to focus on that criterion since the length of a given propositional formula after translation into CNF can be easily computed [8]. The idea of producing short CNF formulas from circuits is similar in spirit to very recent preprocessing approaches [15]. There, a plain CNF formula is analyzed for possible occurrences of renamings, and those renamings are undone if undoing them decreases formula size.

It is interesting to note that the introduction of definitions for subformulas can drastically affect proof length since it essentially allows to simulate analytic cuts over the defined formulas, i.e., cuts that obey the subformula property, even in calculi that are cut-free or only allow for atomic cuts. Consider, as an example of the latter, the DLL procedure. We can choose to view DLL as a tableau procedure consisting of the unit rule and an atomic cut rule which branches on the value of a variable (a proof in such a tableau is then similar to the notion of the search tree in DLL).

If a subformula ϕ has a definition with a naming variable v_ϕ, we can simulate an analytic cut over ϕ by performing an atomic cut on v_ϕ, although such non-atomic cuts are not possible in the tableau procedure. Even in cut-free proof procedures, such a cut can be simulated by introducing definitions. As an example for the simulation of a cut in a cut-free proof procedure, consider the following rule in the semantic tableau for propositional logic:

$$(\vee) \; \frac{A \vee B}{A \mid B}$$

A definition $v_\phi \Leftrightarrow \phi$ translates into $(\neg v_\phi \vee \phi)$ and $(v_\phi \vee \neg\phi)$ if we rewrite the equivalence. It is then possible to use the above rule on those clauses to simulate a cut over ϕ.

Tseitin, who introduced the idea of renamings in [47], opted for a renaming of all subformulas. This in itself is enough to bring down the growth in formula size during clause form translation to a linear factor. This strategy may in exceptional cases even increase the size of the formula, but can be beneficial if one is interested in short proofs (for the reasons outlined above).

Plaisted and Greenbaum introduced in [37] a translation procedure called *structure preserving*, where all subformulas are renamed except for literals and negations. They show that the definitions in Φ_{def} do not need to be translated fully into clause form in order to maintain satisfiability equivalence. Instead, a definition $I \Leftrightarrow \phi$ can be replaced by the clause forms of $I \Leftarrow \phi$ or $I \Rightarrow \phi$, depending on the structure of the original formula.

In order to give an insight into why not all of the clauses are needed to maintain equisatisfiability, we will look at an example in circuit translation. Consider the constrained circuit shown in Figure 4.2. There, the primary output of the gate g_4 with children g_1, g_2 and g_3 is constrained to the value true. The clauses that define the functional behaviour of g_4 are

$$\text{(i) } (g_4 \vee \neg g_1), \;\; \text{(ii) } (g_4 \vee \neg g_2), \;\; \text{(iii) } (g_4 \vee \neg g_3), \quad \text{and (iv) } (\neg g_4 \vee g_1 \vee g_2 \vee g_3)$$

which is the clausal form of $g_4 \Leftrightarrow (g_1 \vee g_2 \vee g_3)$.

We can partition (i)-(iv) into the clauses (i)-(iii) that encode the situation when g_4 evaluates to true and the clause (iv) that encodes the situation when g_4 evaluates to false.

ψ	$\omega^+(\psi)$	$\omega^-(\psi)$
$\psi_1 \vee \ldots \vee \psi_n$	$\neg R(\psi) \vee \bigvee_{1 \le i \le n} R(\psi_i)$	$\bigwedge_{1 \le i \le n} (\, R(\psi) \vee \neg R(\psi_i)\,)$
$\psi_1 \wedge \ldots \wedge \psi_n$	$\bigwedge_{1 \le i \le n} (\, \neg R(\psi) \vee R(\psi_i)\,)$	$R(\psi) \vee \bigvee_{1 \le i \le n} \neg R(\psi_i)$

Table 4.1: Translation table for a polarity-based translation.

From a satisfiability point of view, we gain no information by adding constraints that force g_4 to true, since it would not conflict with our constraint. On the other hand, those clauses that encode the reasons for $g_4 := \mathbf{F}$ are more interesting since they can be used to derive such a conflict. In other words, to preserve satisfiability equivalence, we need not encode information about assignments that cannot conflict with our constraints.

Depending on the polarity of a subformula in the instance, it may be sufficient to encode only the conditions for the subformula evaluating to either true or false. If we only need to encode the conditions for a subformula ϕ of ψ to evaluate to false (or if $\phi = \psi$), we refer to ϕ as being in *positive polarity* w. r. t. ψ, if we only need to encode the conditions for ϕ to evaluate to true, we say ϕ has *negative polarity* w. r. t. ψ. If both are needed, ϕ has *positive and negative polarity.*

We will now give a formal characterization of such a polarity-based translation process. In order to keep the description simple, we will restrict our definition to Boolean functions using the logical operators to $\{\vee, \wedge, \neg\}$. Of course, this is not really a restriction in expressiveness, since we can define $\phi \Rightarrow \psi$ and $\phi \Leftrightarrow \psi$ operators as shorthand notations for $(\neg\phi \vee \psi)$ and $(\neg\phi \vee \psi) \wedge (\phi \vee \neg\psi)$ respectively (although there may be exponential differences in conciseness to the original representation).

Given a formula ϕ, let S'_ϕ be the set of all subformulas of ϕ, and let $S_\phi = S'_\phi \cup \{\phi\}$. We now build a directed graph, $G = (S_\phi, E)$, where $(\psi, \psi') \in E$ iff ψ' is an immediate subformula of ψ. G is a directed acyclic graph with a single root node ϕ. For a path P in G, we define the negation count $\mathrm{negC}(P)$ as the the number of nodes $\neg\psi$ that are contained in the path, where ψ is an arbitrary propositional formula.

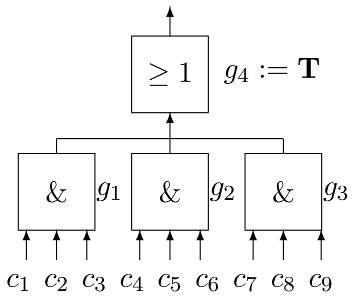
Now, let $l_\phi : S_\phi \mapsto \{+, -, \pm\}$ be a labeling function on nodes, so that $l_\phi(\psi)$ indicates the polarity of ψ w. r. t. ϕ:

$$l_\phi(\psi) = \begin{cases} + & \text{iff for all paths } P \text{ from } \phi \text{ to } \psi, \mathrm{negC}(P) \text{ is even;} \\ - & \text{iff for all paths } P \text{ from } \phi \text{ to } \psi, \mathrm{negC}(P) \text{ is odd;} \\ \pm & \text{otherwise.} \end{cases}$$

Furthermore we define an injective renaming-function R in the following way.

$$R(\psi) = \begin{cases} \neg R(\psi') & \text{iff } \psi = \neg\psi'; \\ \psi & \text{iff } \psi \text{ is a propositional atom;} \\ v & \text{otherwise, where } v \text{ is a variable that does not occur in } \phi. \end{cases}$$

Constrained circuit C:



Propositional formula ϕ of g_4:

$$(c_1 \wedge c_2 \wedge c_3) \vee (c_4 \wedge c_5 \wedge c_6) \vee (c_7 \wedge c_8 \wedge c_9)$$

Formula ϕ translated into CNF:

$$\begin{gathered}
(c_1 \vee c_4 \vee c_7) \wedge (c_1 \vee c_4 \vee c_8) \wedge (c_1 \vee c_4 \vee c_9) \wedge \\
(c_1 \vee c_5 \vee c_7) \wedge (c_1 \vee c_5 \vee c_8) \wedge (c_1 \vee c_5 \vee c_9) \wedge \\
(c_1 \vee c_6 \vee c_7) \wedge (c_1 \vee c_6 \vee c_8) \wedge (c_1 \vee c_6 \vee c_9) \wedge \\
(c_2 \vee c_4 \vee c_7) \wedge (c_2 \vee c_4 \vee c_8) \wedge (c_2 \vee c_4 \vee c_9) \wedge \\
(c_2 \vee c_5 \vee c_7) \wedge (c_2 \vee c_5 \vee c_8) \wedge (c_2 \vee c_5 \vee c_9) \wedge \\
(c_2 \vee c_6 \vee c_7) \wedge (c_2 \vee c_6 \vee c_8) \wedge (c_2 \vee c_6 \vee c_9) \wedge \\
(c_3 \vee c_4 \vee c_7) \wedge (c_3 \vee c_4 \vee c_8) \wedge (c_3 \vee c_4 \vee c_9) \wedge \\
(c_3 \vee c_5 \vee c_7) \wedge (c_3 \vee c_5 \vee c_8) \wedge (c_3 \vee c_5 \vee c_9) \wedge \\
(c_3 \vee c_6 \vee c_7) \wedge (c_3 \vee c_6 \vee c_8) \wedge (c_3 \vee c_6 \vee c_9)
\end{gathered}$$

Tseitin-transformed satisfiability-equivalent formula ϕ_{aux} using auxiliary variables:

$$\begin{gathered}
g_4 \wedge\ (g_1 \Leftrightarrow (c_1 \wedge c_2 \wedge c_3)) \wedge (g_2 \Leftrightarrow (c_4 \wedge c_5 \wedge c_6)) \\
\wedge (g_3 \Leftrightarrow (c_7 \wedge c_8 \wedge c_9)) \wedge (g_4 \Leftrightarrow (g_1 \vee g_2 \vee g_3))
\end{gathered}$$

Formula ϕ_{aux} translated into CNF:

$$\begin{gathered}
g_4 \wedge (\neg g_1 \vee c_1) \wedge (\neg g_1 \vee c_2) \wedge (\neg g_1 \vee c_3) \wedge (g_1 \vee \neg c_1 \vee \neg c_2 \vee \neg c_3) \wedge \\
(\neg g_2 \vee c_4) \wedge (\neg g_2 \vee c_5) \wedge (\neg g_1 \vee c_6) \wedge (g_2 \vee \neg c_4 \vee \neg c_5 \vee \neg c_6) \wedge \\
(\neg g_3 \vee c_7) \wedge (\neg g_3 \vee c_8) \wedge (\neg g_3 \vee c_9) \wedge (g_3 \vee \neg c_7 \vee \neg c_8 \vee \neg c_9) \wedge \\
(g_4 \vee \neg g_1) \wedge (g_4 \vee \neg g_2) \wedge (g_4 \vee \neg g_3) \wedge (\neg g_4 \vee g_1 \vee g_2 \vee g_3)
\end{gathered}$$

Figure 4.2: Example for a circuit-to-CNF translation.

Using the Table 4.1, we can now build the final formula ϕ_{trans},

$$\begin{aligned}\phi_{\text{trans}} &= R(\phi) \wedge \\ & \bigwedge\{\, \omega^+(\psi) \mid \psi \in S_\phi : l_\phi(\psi) = +\} \wedge \\ & \bigwedge\{\, \omega^-(\psi) \mid \psi \in S_\phi : l_\phi(\psi) = -\} \wedge \\ & \bigwedge\{\, \omega^+(\psi) \wedge \omega^-(\psi) \mid \psi \in S_\phi : l_\phi(\psi) = \pm\}.\end{aligned}$$

4.1.2 Producing Short Clause-Forms

As has been stated before, the correspondence between short CNF formulas and shorter running times on SAT solvers is not a clear one. Many small problems are very hard, while many large problems are easy to solve. In DLL, a longer CNF may allow shorter proofs, and, indeed, the whole idea of clause learning is to enrich the formula with redundant information in order to navigate the search space more efficiently. Still, CNF length is trivial to compute and it can be a useful heuristic. This is indicated by the high success of preprocessing techniques that reduce the size of the CNF in state-of-the-art SAT solvers [15].

As we have seen, renamings can be used to keep the size of the translated CNF formula linear in the size of the old formula. In some cases, renamings may actually increase clause size compared with a direct translation. Consider, as a trivial example, a circuit consisting of a single gate $g = \text{or}(c_1, \ldots, c_n)$, with its output constrained to true. Tseitin's translation would produce the following clauses

$$(\neg g \vee c_1 \vee \ldots \vee c_n) \wedge (g \vee \neg c_1) \wedge \ldots \wedge (g \vee \neg c_n),$$

while a direct translation could be formulated simply as

$$(c_1 \vee \ldots \vee c_n).$$

In Boy de la Tour [8], an algorithm is introduced which chooses a renaming that is optimal in the sense that no other renaming produces fewer clauses. Starting from the original formula, the algorithm considers all subformulas in a descending order and introduces definitions and naming variables for them only if it does not increase the number of clauses in the translation.

In addition to reducing the number of clauses by choosing a proper renaming, we can try to reduce the number of clauses by incorporating circuit-specific information. In the technique presented in Velev [48], which is built upon a renaming-based approach, trees of If-Then-Else (ITE) operators are identified in the circuit and translated into clause form without the use of renamings.

An ITE operator selects between two signals based on the value of a third control input. We can therefore consider it to implement a two-to-one multiplexer or a controlled switch. If we now consider a binary tree of ITE operators which are connected through their non-control inputs, we have an n-to-one multiplexer, where n is the number of leaves on the binary tree. Instead of translating this arrangement by choosing a proper set of renamings and then flattening it into CNF, we can simply introduce two clauses for each leaf of the tree, which encode the configuration of the control inputs which would propagate a given signal to the output o of the topologically highest ITE operator. In a further refinement, non-ITE gates which feed directly into the ITE tree can also be factored into the produced clauses.

More formally, for each non-control input variable s to the ITE tree, we add the implications $(s \wedge \mathit{selected}(s)) \Rightarrow o$ and $(\neg s \wedge \mathit{selected}(s)) \Rightarrow \neg o$, where $\mathit{selected}(s)$ is the conjunction of control input literals that cause s to be selected. ITE trees are only translated using this technique if no intermediate outputs are used elsewhere in the circuit.

4.1.3 Enriching CNF with Deduction Shortcuts

So far, the motivation behind the discussed translation strategies has been to minimize the number of clauses and variables, ultimately, to increase the efficiency of a SAT solver working on the resulting CNF. Another possible approach is to speed up the search process by introducing new clauses and variables which encode additional information about the circuit. These clauses may act as implication shortcuts, allowing the solver to effectively prune parts of the search space during the solving process.

The method described in Velev [49] aims to speed up the solving process by introducing new variables and clauses which describe the observability of subcircuits that have one primary output. Given a circuit B, a subcircuit C of B with one primary output, and a partial assignment τ on the gates in $\mathcal{G}_B \backslash \mathcal{G}_C$, we will call C unobservable at the primary outputs of B under τ if the value of the primary output of C cannot influence the value of any primary output of B. As it is of no concern which value the primary output of C takes, the whole subcircuit can be safely ignored in the search process.

Signal unobservability has already been exploited in hybrid solvers such as the one described in Gupta et al. [21], which incorporate circuit-specific optimizations into standard CNF solvers. The method discussed here differs insofar as it does not propose a SAT algorithm which explicitly takes into account circuit observability information, but encodes the information instead into CNF clauses and unobservability variables. Consequently, standard CNF-based SAT procedure automatically makes use of this information.

Some chosen subcircuits are associated with unobservability variables. If an unobservability variable is set to true, the corresponding subcircuit's output is not observable, if the variable is set to false, it may be observable, but not necessarily so. The conditions for unobservability are encoded in clause form and all clauses that partially describe a subcircuit are extended with the unobservability variable of that subcircuit. When the conditions for unobservability are met, the corresponding unobservability variable is set to true, and all clauses that describe the unobservable part of the circuit are automatically satisfied. This makes decisions on variables which only occur in the unobservable part unnecessary. Furthermore, it is possible to explicitly consider the cases where a given subcircuit is unobservable or observable by making a decision on an unobservability variable.

To determine the unobservability of a subcircuit, new clauses have to be introduced, which drive the unobservability variable to true or false. Given a subcircuit with output signal o_C and associated unobservability variable u_C, three types of clauses, which capture different conditions for observability and unobservability, are therefore introduced.

Firstly, if, starting from o_C, there exists a sequence P of forward-connected gates whose outputs do not branch, then for each gate on P, unobservability conditions are added. Such a path P is also called a fanout-free partial path. As the signal o_C can only reach the primary output via P, it becomes unobservable if it becomes unobservable on a gate on P. Each of the added clauses encodes an implication describing a condition for unobservability at one of the gates on P, e.g., for an *and* gate with inputs $a_1, \ldots, a_n$ and one additional input $a_P \in P$, we add the clause form of each implication $a_i \Rightarrow u_C$ for $i = 1, \ldots, n$. For an *or* gate with inputs $b_1, \ldots, b_n$ and $b_P \in P$ we add the clause form of $\neg b_i \Rightarrow u_C$ for $i = 1, \ldots, n$. This encodes our knowledge that one false input in an *and* gate and one true input in an *or* gate fully determine the gate output and thus make all other inputs unobservable.

Secondly, if subcircuits on all paths from o_C to the primary output have been determined to be unobservable, then C is also unobservable. More prosaically, we can say that if all possible paths from C to the primary output are blocked by unobservable parts of the circuit, the output value of C is also blocked. For instance, if there exist n paths from o_C to the primary output, where on each path P_i $(1 \leq i \leq n)$ a subcircuit with an associated unobservability variable u_i exists, then the following clause is added:

$$\neg u_1 \vee \neg u_2 \vee \cdots \vee \neg u_n \vee u_C$$

Thus, when all paths towards the output become unobservable, u_C is set to 1. If multiple subcircuits with associated unobservability variables exists on a path P_i, the one that is nearest to o_C is used in the clause.

Thirdly and lastly, another type of clause is necessary which forces all unobservability variables to zero if unobservability could not explicitly be proven. This is necessary in order to maintain equisatisfiability to the original CNF formula. If we would not add this last type of clause, the unobservability variables would only occur as positive literals in the whole CNF formula, which would in turn lead all clauses that contain an unobservability variable to be trivially satisfied by setting that variable to true.

Overall, the number of added clauses for a circuit substructure C is equal to

$$p + \sum_{i=0}^{g} (inp_i - 1)$$

where g is the number of gates on the fanout-free partial path P which may cause C to be unobservable, inp_i is the number of inputs of the i-th gate on P, and p is the number of possible paths from C to the primary circuit output. The question of how much additional effort is necessitated by the identification of suitable circuit substructures needs further consideration. An illustration of this techniques is given in Figure 4.3.

The method is compared with a set of other non-trivial translation techniques [50] which also account for observability information. This set also includes the translation techniques from Velev [48], which were described above. In the evaluated test cases, the method yields a small speedup of up to two compared to the previous best.

4.2 Combining Circuit-SAT and CNF-SAT

CNF-based solvers have some advantages which make them well-suited as a basis for circuit-based SAT solvers: They are very common and have seen a tremendous increase in efficiency over the last few years for practical problems [44]. Furthermore, algorithms for CNF-SAT are a well-established research area, and new improvements can easily be applied to target applications.

On the other hand, topological and structural circuit information is lost during the circuit-to-CNF translation, which may be of crucial importance in the solving process. Also, as some applications make repeated runs on the same circuit structures necessary, the translation process can take up a significant amount of runtime [48].

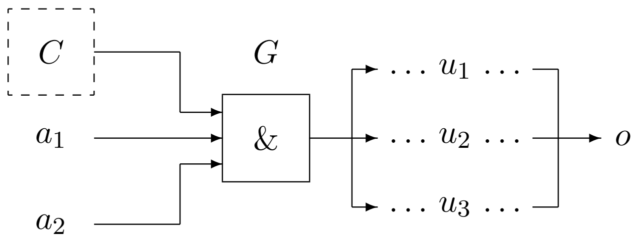
In the previous section, approaches were discussed which aim to encode circuit information directly into the CNF formula. Here, dedicated SAT solvers will be presented, which explicitly consider this information in the solving process by combining CNF-based solving procedures with circuit representations.

4.2.1 Introducing Circuit Information into CNF-SAT

In Silva et al. [38], a generic SAT solving algorithm is supplemented with a layer that handles circuit specific information such as information about children and parent gates and justification information. Remember that a gate is justified in a partial assignment if its assigned output value can be deduced from its current input assignments alone, e.g., a $\mathbf{F}(\mathbf{T})$ at the output of an *and* (*or*) gate can be justified by a value of $\mathbf{F}(\mathbf{T})$ at any one of the inputs.

For each gate, the number of assigned true and false inputs as well as the number of necessary true and false inputs for justifications are stored. An *or* gate, for instance, is justified with one true input or with all inputs set to false, while an *xor* gate needs all inputs set in both cases. A justification frontier is maintained, which stores those gates still in need of justification, i.e., whose output variable is set to a value which does not necessarily result from its input values. The justification process works backwards from the primary output (or other constrained variables) by iteratively trying to find values for gate inputs which are sufficient to explain the output. The circuit SAT instance is satisfied, if, at a point during search, no further gates need to be justified, and unsatisfiable if justification of the constrained gates is unsuccessful, i.e., if the algorithm finds no assignments that could produce the chosen output.

This process produces a partial assignment in the case of a satisfiable instance. This assignment can easily be extended to a total assignment by setting the unassigned input gates to arbitrary values and propagating them upwards.



For the subcircuit C, an unobservability variable u_C and a number of clauses are added which encode the conditions for unobservability. In this case, the only gate for which clauses are added is G as there is no longer fanout-free path from C to the primary output.

C becomes unobservable at gate G if a_1 or a_2 is false

$$a_1 \vee u_C \qquad a_2 \vee u_C$$

C becomes unobservable if all paths towards the primary output lead through unobservable subcircuits

$$\neg u_1 \vee \neg u_2 \vee \neg u_3 \vee u_C$$

C may be observable if there is at least one observable path to the output and gate G does not cause C to become unobservable

$$\neg a_1 \vee \neg a_2 \vee u_1 \vee \neg u_C$$

$$\neg a_1 \vee \neg a_2 \vee u_2 \vee \neg u_C$$

$$\neg a_1 \vee \neg a_2 \vee u_3 \vee \neg u_C$$

Figure 4.3: Example for the introduction of unobservability variables and clauses as presented in Velev [49].

In order to make use of the structural information in the SAT procedure, node justification information is updated during BCP. The check for satisfiability, which is usually done by asserting that all clauses are satisfied, simply tests if the justification frontier is empty. Furthermore, the information about children gates can be used to implement efficient structure-based heuristics for decision variable selection.

Experiments were performed by adding the proposed circuit layer on top of the Grasp SAT solver [33] in a solver named CGrasp and comparing the two algorithms for the tasks of test pattern generation and circuit delay computation, both of which seem to indicate good overall speedups.

4.2.2 Circuit-SAT with Clause Learning

The SAT solver proposed in Ganai et al. [18], on the other hand, does not translate the input circuit into CNF, but directly uses an *AND-Inverter Graph* (AIG) as a circuit-based representation. An AIG is a directed acyclic graph representing a propositional formula which only uses 2-input *and* primitives and inverter attributes on edges. CNF-based representation is still employed for learned clauses.

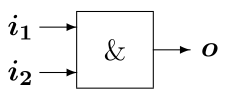
The circuit-based representation of the input enables efficient Boolean constraint propagation (BCP). While translating a gate into CNF usually produces multiple clauses and therefore takes multiple BCP steps, the circuit-based BCP can be done efficiently in one step by using a lookup table [29]. In the solving process, each gate's status is given by three values, two inputs and one output, where each value can either be true, false or unassigned. When considering the implications of a gate assignment, the lookup table is queried for the current configuration, which either returns the implied assignments, or the occurrence of a conflict. In Figure 4.4, an illustration is shown.

Learned clauses obtained by conflict analysis are typically very large. Including them as 2-input gate trees can increase the size of the circuit significantly, which leads to a high number of required implications during BCP, and therefore voids all benefits of the circuit-based representation. For this reason, learned clauses are represented as CNF formulas.

The hybrid approach that was just described is also the basis for the solver presented in Lu et al. [30]. Just as in Ganai et al. [18], the input circuit is represented as an AIG, while learned clauses are included as CNF formulas. On top of this idea, a simulation-based technique is developed which efficiently identifies signal correlation, and two different heuristics are presented which make use of the correlation information to speed up the solving process.

Three types of correlation between two signals s_i and s_j are identified: $s_i = s_j$, $s_i \neq s_j$, and constant correlation where either $s_i = 0$ or $s_i = 1$. The identification of these relationships is performed using simulations with randomly assigned inputs. Signals are assumed to be correlated if they consistently show the same values or if they consistently show different values during a number of these simulation runs.

The following hashing method is proposed in order to identify groups of correlated signals in constant time after simulation. A random assignment is chosen for the input variables of the circuit, the values of the other gates are then determined via BCP. This



gate assignments			implied assignments		
i_1	i_2	o	i_1	i_2	o
1	1	?	1	1	1
0	x	?	0	x	0
1	?	0	1	0	0
?	?	1	1	1	1
0	x	1	CONFLICT		
1	1	0	CONFLICT		
...			...		

Figure 4.4: Examples for BCP based on table look-up in an AND gate. The left subtable lists possible gate configurations, the right subtable lists all implied assignments. Question marks represent unassigned inputs. Variables represent don't-care inputs or arbitrary values.

step is repeated 32 times and each gate is associated with a word (32 bits) which stores the results for the runs, so that the first bit represents the gate value in the first run, the second bit for the second run, and so on. Two signals are assumed to be correlated if their 32 bit result values are identical or bit-wise inversions of each other. Hashing is used to determine these associations efficiently.

After groups of correlated signals have been found, another 32 simulation runs are performed, and the correlated-signal groups are updated. This process is continued until enough correlations have been found.

Two heuristics are proposed which make use of correlation between signals: Implicit learning and explicit learning. Implicit learning is a variable selection heuristic, in which correlated signals are grouped together during the variable selection process, so that the chance of immediately generating a conflict is increased. If, for example, two signals s_i and s_j show an equivalence correlation, so that $s_i = s_j$ is most likely true, and $s_i = 1$ has been the most recent decision, then $s_j = 0$ will be the next decision. As $s_i = 1$ and $s_j = 0$ are unlikely to occur together this selection will probably lead to a conflict soon.

In explicit learning, several smaller SAT instances are generated from subcircuits of the actual circuit before the full SAT problem is tackled. While solving the subproblems, smaller learned clauses are generated, which can be used to prune the search space of the larger problems more effectively. As mainly the learned clauses are of interest, and not whether the subproblem is actually satisfiable, runs may be stopped early.

Similar to implicit learning, the subproblems are generated in a way that maximizes the chance for conflicts. For instance, for a signal s_j that has a constant correlation

$s_j = 0$, a subproblem $s_j = 1$ could be generated. Furthermore, if a signal s_k has a correlation with a topologically lower signal s_l so that $s_k \neq s_l$ is likely, then two subproblems are generated: $s_k = 1, s_l = 1$ and $s_k = 0, s_j = 0$. All generated subproblems are solved in topological order, from lowest to highest, so that the learned clauses from a topologically lower subproblem can be used during the solving process of a topologically higher subproblem. Another advantage of solving the topologically lower subproblems first is that fewer variables are involved and therefore the learned clauses tend to be smaller.

In Lu et al. [30], implicit learning is shown to speed up the solving process by an order of magnitude on some instances. Explicit learning speeds up the solver by another order of magnitude compared to implicit learning, but combining implicit and explicit learning does not lead to further speedups.

4.3 Circuit-based Approaches

As circuit-based approaches we classify those SAT-techniques which mainly rely on direct circuit representations such as AIGs and do not include a CNF-based representation.

The solver in Kuehlmann et al. [29] combines an AIG-based SAT procedure with two techniques which iteratively simplify the circuit: BDD sweeping [28] and local structural transformations. Advanced search techniques which are common in modern CNF-based solvers, such as non-chronological backtracking and conflict-driven learning, are adapted to the AIG-representation and implemented in the solving procedure. In addition, a static learning scheme is introduced, which adds implication shortcuts to the graph during graph construction or modification.

The local structural transformations are automatically applied during graph build-up or modification. During the construction of a new vertex, a functional hashing algorithm is used in order to find isomorphic vertices and thus reduces circuit redundancy. If the table lookup does not find an isomorphic existing vertex, the current vertex and its two children are converted into a canonical format and the functional hashing scheme is applied again to the resulting 4-input substructure. The conversion to the canonical format ensures that logically equivalent arrangements of a gate and its input gates are represented identically in the circuit, while the hashing algorithm eliminates the structural redundancy that results.

The hash-table that is created in this step is also used for a static learning technique. The idea is similar to conflict-driven learning, where implication shortcuts are added dynamically to avoid reentering certain parts of the search-space that invariably lead to conflicts. Here, implication shortcuts are integrated into the graph according to static rules to avoid some of those parts of the search-space altogether. The hash-table is used to identify certain pairs of gates that typically occur very often and whose evaluation can be sped up in some cases by adding implication shortcuts.

The SAT solver implements a basic DLL-algorithm that operates directly on the AIG. The procedure is started with an initial assignment and exhaustively tries to find consistent assignments for all other vertices by propagating Boolean constraints and justifying AIG vertices by case splits. As atomic building blocs of an AIG are 2-input structures, Boolean constraint propagation can be handled efficiently by a table lookup. Given the current assigned values of an AIG vertex and its inputs, the lookup table determines whether a new assignment is implied, a conflict has occurred, or a case split is necessary for node justification. If, after applying BCP, the justification queue becomes empty, the circuit-SAT instance is satisfiable.

As already mentioned, the solver also implements non-chronological backtracking and conflict-driven learning. In order to use these techniques, it is necessary to monitor the causes of variable assignments, which is typically handled by an implication graph in CNF-based solvers. The AIG SAT solver substitutes the implication graph with bit-vectors, where each bit represents a case-split assignment. This bit-vector is propagated along the implication path during BCP using bit-wise *or* operations, e.g., if two assigned inputs cause the output to be assigned, the bit-vectors of the inputs are propagated to the output by combining the input bit-vectors via bit-wise *or* operations. When a conflict is derived in a vertex of the AIG, the conflicting value carries the case assignments that are responsible in its bit-vector. After all assignments on a decision level have been proven to lead to conflicts, the backtracking level can be determined by choosing the lowest decision level that was involved in the conflict. Information learned from these conflicts are added as additional gate structures to the AIG, which essentially act as implication short-cuts. Local structural transformations are automatically applied during the creation of these vertices, which can lead to further simplifications of the graph.

After local structural transformation and the AIG SAT algorithm, the last solving strategy used in Kuehlmann et al. [29] is BDD sweeping [28], which searches for functionally identical or complementary vertices in the AIG graph by building and hashing their BDDs. Since BDDs are a canonical format (if a variable ordering is imposed and reduction rules are applied), they allow for structural equivalency checking. Equivalent or complementary vertices are then merged, which again triggers the application of local structural transformation.

The BDD sweeping and AIG SAT algorithm are applied iteratively with limits on the maximum number of backtracks and maximum BDD size. Between iterations, these resource limits are increased. Intermediate SAT results and BDDs are stored, so that the algorithms can pick up from where they hit their resource limit in the previous iteration.

All approaches that have been presented up to this point are based on derivatives of the basic DLL procedure in one way or another. The method described in Junttila and Niemelä [26], which will be the focus of Chapter 5, employs a different strategy and uses a tableau calculus for Boolean circuits. With six different basic gate types, the representation of the input is rather complex compared to AIG-based solvers and therefore allows to express circuit structures more compactly.

Traditionally, tableau-calculi that are used in automatic deduction are cut-free, i.e., they do not contain applications of a cut rule. This keeps the calculus analytic and therefore avoids having to draw intermediate formulas "out of thin air". The tableau calculus presented in Junttila and Niemelä [26] follows another approach and contains

$$\frac{v \in V}{\mathbf{T}v \mid \mathbf{F}v}$$

Figure 4.5: The explicit cut rule in Junttila and Niemelä [26].

an atomic cut rule (shown in Figure 4.5), which is its only branching rule.

The reason for this approach is that cut-free tableau calculi suffer from some anomalies, including computational ones (e.g., [1; 17]). For some special classes of formulas, cut-free tableaus perform worse than simple enumeration of all cases, for example. The use of a cut rule similar to the one in Figure 4.5 can solve these problems. The calculus is still analytic, as the application of the atomic cut rule obeys the subformula principle if the application of the cut rule is restricted to variables that occur in the formula to be proved. In a practical implementation, the atomic cut rule is only applied if no other rule is applicable and only invoked on variables which have not yet been assigned a value. This is similar to a decision in a DLL-based solver, which is made only after no new assignments can be deduced via BCP.

In addition to the core tableau calculus, which is complete and therefore sufficient for determining satisfiability in a circuit, three distinct strategies are used to speed up the solving process. An additional set of tableau rules is added, which are not necessary for completeness, but represent deduction shortcuts that improve solving efficiency. A one-step lookahead search heuristic is employed: If on a branch the introduction of an expression $\mathbf{T}v$ ($\mathbf{F}v$) allows to deduce a clash on the branch (a pair of expressions $\mathbf{T}w$ and $\mathbf{F}w$) without using the cut rule, then $\mathbf{F}v$ ($\mathbf{T}v$) is deduced.

Finally, rewriting rules are applied to the constrained circuit before the solving process starts. These include the sharing of common subexpressions, a cone of influence reduction, which removes those parts of the circuit that cannot influence constrained gates, and the removal of some gate inputs whose value has been determined through the propagation of initial constraints. For example, true inputs to *and* gates are removed.

BCSat, an experimental implementation of the tableau-based procedure presented in Junttila and Niemelä [26], is compared with circuit-based as well as CNF-based solvers on examples from bounded model-checking. No clear conclusions can be drawn from the results. In most cases the solver is faster than CGrasp [38] by between one and two orders of magnitude, but in some unsatisfiable instances BCSat is significantly slower. Comparison between BCSat and CNF-based solvers is even less conclusive. Overall, BCSat's tableau technique seems to be systematically on-par with traditional SAT solvers and could therefore be an interesting basis for future research in circuit satisfiability. Chapter 5 will explore how the BC-Tableau can be combined with many of the techniques in CNF-SAT.

Chapter 5

Implementing an Extended Tableau-Based SAT Solver

In this chapter, we will build upon the work presented in Junttila and Niemelä [26] and present a tableau-based SAT solver which we extend with many of the techniques found in CNF-SAT. We will present the idea that BC-tableau-based SAT solving is a generalized form of DLL, and can therefore be extended easily with many of the advances made in CNF-based SAT solving. As a proof-of-concept, we have implemented BattleAx^3 ("BattleAx Cube" being an anagram of "BC Tableau Ext."), a BC-based SAT solver that performs learning and non-chronological backtracking, and we give an evaluation of some popular techniques in CNF-based SAT solving as applied to a tableau-based solver.

In the course of writing this thesis, the author was made aware of the then unfinished work of Drechsler et al. [14], which presents some similar ideas of viewing the BC tableau as generalized DLL and gives a framework for including backjumping and clause-learning, but does not give a prototypical implementation. At this point, the work on BattleAx^3 was already finished. For completeness, we reference the work of Junttila and Niemelä [26] in the following chapter, but the ideas that pertain to BC-as-DLL and the introduction of advances from CNF-based SAT were still independently explored by the author.

5.1 Tableau-Based SAT Solving

The BC tableau is a tableau calculus for determining the satisfiability of a formula. It was introduced in Junttila and Niemelä [26] and presented in more detail in Drechsler et al. [14] and consists of a number of straightforward deduction rules for gates, together with an atomic cut rule that is similar to decisions in the DLL procedure. In Junttila and Niemelä [26], an experimental implementation of the BC tableau is presented. In this section, we will expand upon the work presented there by showing how the basic BC tableau can be combined with non-chronological backtracking and learning in our prototypical implementation BattleAx^3, and by giving a thorough evaluation of many of the techniques found in CNF-based SAT solving as applied to circuit structures.

5.1.1 The BC Tableau

A *BC tableau* is a directed tree where the vertices are occurrences of statements, and each statement is either a gate equation of the form $g = f_g(c_1, \ldots, c_n)$, or an occurrence of an assignment of form $g := \mathbf{T}$ or $g := \mathbf{F}$. We refer to the vertices in a BC tableau as *tableau nodes.* We call a node of form $g := \mathbf{T}$ or $g := \mathbf{F}$ an *assignment node.* All edges in a tableau are oriented away from a single *root node.* A *leaf* is a node which has no outgoing edge. A path from the root node to a leaf is called a *branch.* Before we give a full formal characterization of the BC tableau, we present a first characterization of tableau rules.

We will use the notation $n_1 \to \ldots \to n_k$ to denote a branch (V, E), where $V = \{n_1, \ldots, n_k\}$ and $E = \{(n_1, n_2), (n_2, n_3), \ldots (n_{k-1}, n_k)\}$.

First, let $\mathcal{R}$ be the *set of BC tableau rules.* Each tableau rule in $\mathcal{R}$ is a pair of the form (P, Q) where $P = \{p_1, \ldots, p_n\}$ is a set of tableau nodes and $Q = \{Q_1, \ldots, Q_j\}$ where $Q_1 = \{q_{1,1}, \ldots, q_{1,l_1}\}, \ldots, Q_j = \{q_{j,1}, \ldots, q_{j,l_j}\}$ are sets of tableau nodes. We use the alternative notation

$$\begin{array}{c|c|c}
\multicolumn{3}{c}{p_1} \\
\multicolumn{3}{c}{\vdots} \\
\multicolumn{3}{c}{p_n} \\
\hline
q_{1,1} & \cdots & q_{j,1} \\
\vdots & \vdots & \vdots \\
q_{1,l_1} & \cdots & q_{j,l_j}
\end{array}$$

to denote the pair (P, Q). The set of tableau rules will be characterized in detail later, the basic rules can be seen in Figure 5.1.

A rule is *applicable* on a branch $B = (b_1, \ldots, b_k)$ iff $P \subseteq \{b_1, \ldots, b_k\}$. We say a tableau T is a result of *applying* (P, Q) on a second tableau T' iff (P, Q) is applicable on a branch $B = (b_1, \ldots, b_k)$ in T', and T can be constructed from T' by appending the subtableaus $q_{1,1} \to \ldots \to q_{1,l_1}$ to $q_{j,1} \to \ldots \to q_{j,l_j}$ to the leaf node b_k with edges $(b_k, q_{1,1})$ to $(b_k, q_{j,1})$. The BC tableau rules will be presented in detail later.

We call a BC tableau *regular* iff no branch exists which contains two distinct occurrences of the same statement. We call a tableau T the result of *partially regularizing* T' iff T is the result of replacing a subgraph $x \to y \to z$ in T' with $x \to z$ where there is a node y' on the path from the root node to y in T' with y and y' being two distinct occurrences of the same statement.

We can now give a formal characterization of well-formed BC tableaus.

Definition 21. *Let (C, τ) be a constrained circuit with $C = (\mathcal{G}, \mathcal{E})$ and*

$$\mathcal{E} = \{g_1 := f_{g_1}(c_{1,1}, \ldots c_{1,k_1}), \ldots, g_n := f_{g_n}(c_{n,1}, \ldots c_{n,k_n})\}$$

and let

$$\tau = \{v_1 := X_1, \ldots, v_r := X_r\} \textit{ with } X_i \in \{\mathbf{T}, \mathbf{F}\} \textit{ for } 1 \leq i \leq r.$$

Then the set $\mathcal{T}_{(C,\tau)}$ of well-formed BC tableaus of (C,τ) is inductively defined as follows:

[B] Let T be the tableau

$$\begin{aligned}(\, g_1 := f_{g_1}(c_{1,1},\ldots c_{1,k_1})\,) \;\; &\rightarrow \ldots \rightarrow \;\; (\, g_n := f_{g_n}(c_{n,1},\ldots c_{n,k_n})\,) \\ \rightarrow v_1 := X_1 \;\; &\rightarrow \ldots \rightarrow \;\; v_r := X_r\end{aligned}$$

We refer to this tableau as the root tableau *of (C,τ). Then $T \in \mathcal{T}_{(C,\tau)}$.*

[S1] Let T', T be BC tableaus so that $T' \in \mathcal{T}_{(C,\tau)}$ and T is the result of applying a tableau rule on T', then $T \in \mathcal{T}_{(C,\tau)}$.

[S2] Let T', T be BC tableaus so that $T' \in \mathcal{T}_{(C,\tau)}$ and T is the result of partially regularizing T', then $T \in \mathcal{T}_{(C,\tau)}$.

We can make the notion of a BC tableau of a constrained circuit (C,τ) more intuitive by giving a characterization of a tableau procedure which iteratively builds up a BC tableau. The tableau procedure is initialized with a single branch, containing all the equations of C and all assignments in τ. For easier readability, we will use $\mathbf{T}v$ to denote that $v := \mathbf{T}$ and $\mathbf{F}v$ to denote that $v := \mathbf{F}$. Then the tableau rules are applied to deduce new tableau nodes, either in the form of a straightforward deduction, or by application of the cut rule, which splits the current branch into two new branches.

We call a branch that contains a pair $\mathbf{T}v, \mathbf{F}v$ *contradictory*, otherwise we call it *open*. A branch B is *complete* iff either B is contradictory or, for every variable v, either $\mathbf{T}v$ or $\mathbf{F}v$ is in B. A tableau is *finished* iff all branches are contradictory or at least one branch is complete and open. A tableau is *closed* iff all branches are contradictory.

A constrained circuit instance (C,τ) is unsatisfiable iff there exists a closed BC tableau for (C,τ). Such an instance is satisfiable, on the other hand, iff there exists a finished BC tableau for (C,τ) that is not closed. In this case, each completed and open branch in such a tableau yields a satisfying assignment.

The basic tableau rules are shown in Figure 5.1. In order to stay consistent with our formal definition, these have to be thought of as schemata for tableau rules, with a concrete tableau rule being an instance of that schema. The set $\mathcal{R}$ is then the set of all instances of the presented rule schemata. To simplify this discussion, we will still refer to these schemata as rules when, in fact, we refer to their instances.

The size of a tableau depends essentially on the number of cuts (and therefore branches) that are necessary. In order to reduce the number of necessary branches, the basic tableau rules are supplemented with additional deduction rules, shown in Figure 5.2. All rules except the cut rule will be referred to as *deterministic deduction rules*. A variable assignment $\mathbf{T}v$ or $\mathbf{F}v$ can be *deduced* on a branch, if it can be added to the branch by application of the deterministic deduction rules only. For an example of a tableau using basic and additional tableau rules see Figure 5.4.

Additionally, the BC1.0 input format [24] that is accepted by the experimental implementation in Junttila and Niemelä [26] includes *ite* (*if-then-else*) gates and *card* gates. An *ite* gate acts as a two-to-one multiplexer, a *card* gate has associated values x and y,

$$\frac{v \in \mathcal{B}}{\mathbf{T}v \;\big|\; \mathbf{F}v}$$

(1) Atomic cut rule

$$\frac{v = \mathrm{true}()}{\mathbf{T}v} \qquad \frac{v = \mathrm{false}()}{\mathbf{F}v}$$

(2-3) Constant rules

$$\frac{\begin{array}{c} v = \mathrm{not}(v_1) \\ \mathbf{F}v_1 \end{array}}{\mathbf{T}v} \qquad \frac{\begin{array}{c} v = \mathrm{not}(v_1) \\ \mathbf{T}v_1 \end{array}}{\mathbf{F}v}$$

(4-5) Negation rules

$$\frac{\begin{array}{c} v = \mathrm{or}(v_1, \ldots, v_k) \\ \mathbf{F}v_1, \ldots, \mathbf{F}v_k \end{array}}{\mathbf{F}v} \qquad \frac{\begin{array}{c} v = \mathrm{and}(v_1, \ldots, v_k) \\ \mathbf{T}v_1, \ldots, \mathbf{T}v_k \end{array}}{\mathbf{T}v} \qquad \frac{\begin{array}{c} v = \mathrm{or}(v_1, \ldots, v_k) \\ \mathbf{T}v_i, i \in \{1, \ldots, k\} \end{array}}{\mathbf{T}v} \qquad \frac{\begin{array}{c} v = \mathrm{and}(v_1, \ldots, v_k) \\ \mathbf{F}v_i, i \in \{1, \ldots, k\} \end{array}}{\mathbf{F}v}$$

(6-9) "Up" rules for *or* gates and *and* gates

$$\frac{\begin{array}{c} v = \mathrm{equiv}(v_1, \ldots, v_k) \\ \mathbf{T}v_1, \ldots, \mathbf{T}v_k \end{array}}{\mathbf{T}v} \qquad \frac{\begin{array}{c} v = \mathrm{equiv}(v_1, \ldots, v_k) \\ \mathbf{F}v_1, \ldots, \mathbf{F}v_k \end{array}}{\mathbf{T}v} \qquad \frac{\begin{array}{c} v = \mathrm{equiv}(v_1, \ldots, v_k) \\ \mathbf{T}v_i, 1 \leq i \leq k \\ \mathbf{F}v_j, 1 \leq j \leq k \end{array}}{\mathbf{F}v}$$

(10-12) "Up" rules for *equiv* gates

$$\frac{\begin{array}{c} v = \mathrm{even}(v_1, \ldots, v_k) \\ \mathbf{T}v_1, \ldots, \mathbf{T}v_j,\ j \text{ is even} \\ \mathbf{F}v_{j+1}, \ldots, \mathbf{F}v_k \end{array}}{\mathbf{T}v} \qquad \frac{\begin{array}{c} v = \mathrm{even}(v_1, \ldots, v_k) \\ \mathbf{T}v_1, \ldots, \mathbf{T}v_j,\ j \text{ is odd} \\ \mathbf{F}v_{j+1}, \ldots, \mathbf{F}v_k \end{array}}{\mathbf{F}v}$$

$$\frac{\begin{array}{c} v = \mathrm{odd}(v_1, \ldots, v_k) \\ \mathbf{T}v_1, \ldots, \mathbf{T}v_j,\ j \text{ is odd} \\ \mathbf{F}v_{j+1}, \ldots, \mathbf{F}v_k \end{array}}{\mathbf{T}v} \qquad \frac{\begin{array}{c} v = \mathrm{odd}(v_1, \ldots, v_k) \\ \mathbf{T}v_1, \ldots, \mathbf{T}v_j,\ j \text{ is even} \\ \mathbf{F}v_{j+1}, \ldots, \mathbf{F}v_k \end{array}}{\mathbf{F}v}$$

(13-16) "Up" rules for *even* gates and *odd* gates

Figure 5.1: BC tableau basic rules. The numbering is left-to-right. Note that for easier readability, the ordering of children given in the rules does not correspond to the ordering in the original equations, i.e., a rule is applicable iff the children in the original equation can be reordered as above.

$$\frac{\begin{array}{c} v = \mathrm{not}(v_1) \\ \mathbf{T}v \end{array}}{\mathbf{F}v_1} \qquad \frac{\begin{array}{c} v = \mathrm{not}(v_1) \\ \mathbf{F}v \end{array}}{\mathbf{T}v_1}$$

(17-18) "Down" rules for *not* gate

$$\frac{\begin{array}{c} v = \mathrm{or}(v_1, \ldots, v_k) \\ \mathbf{F}v \end{array}}{\mathbf{F}v_1, \ldots, \mathbf{F}v_k} \qquad \frac{\begin{array}{c} v = \mathrm{and}(v_1, \ldots, v_k) \\ \mathbf{T}v \end{array}}{\mathbf{T}v_1, \ldots, \mathbf{T}v_k} \qquad \frac{\begin{array}{c} v = \mathrm{equiv}(v_1, \ldots, v_k) \\ \mathbf{T}v_i, i \in \{1, \ldots, k\} \\ \mathbf{T}v \end{array}}{\mathbf{T}v_1, \ldots, \mathbf{T}v_k} \qquad \frac{\begin{array}{c} v = \mathrm{equiv}(v_1, \ldots, v_k) \\ \mathbf{F}v_i, i \in \{1, \ldots, k\} \\ \mathbf{T}v \end{array}}{\mathbf{F}v_1, \ldots, \mathbf{F}v_k}$$

(19-22) "Down" rules for *or* gates, *and* gates and *equiv* gates

$$\frac{\begin{array}{c} v = \mathrm{or}(v_1, \ldots, v_k) \\ \mathbf{F}v_1, \ldots, \mathbf{F}v_{k-1}, \mathbf{T}v \end{array}}{\mathbf{T}v_k} \qquad \frac{\begin{array}{c} v = \mathrm{equiv}(v_1, \ldots, v_k) \\ \mathbf{T}v_1, \ldots, \mathbf{T}v_{k-1}, \mathbf{T}v \end{array}}{\mathbf{T}v_k} \qquad \frac{\begin{array}{c} v = \mathrm{equiv}(v_1, \ldots, v_k) \\ \mathbf{T}v_1, \ldots, \mathbf{T}v_{k-1}, \mathbf{F}v \end{array}}{\mathbf{F}v_k}$$

$$\frac{\begin{array}{c} v = \mathrm{and}(v_1, \ldots, v_k) \\ \mathbf{T}v_1, \ldots, \mathbf{T}v_{k-1}, \mathbf{F}v \end{array}}{\mathbf{F}v_k} \qquad \frac{\begin{array}{c} v = \mathrm{equiv}(v_1, \ldots, v_k) \\ \mathbf{F}v_1, \ldots, \mathbf{F}v_{k-1}, \mathbf{T}v \end{array}}{\mathbf{F}v_k} \qquad \frac{\begin{array}{c} v = \mathrm{equiv}(v_1, \ldots, v_k) \\ \mathbf{F}v_1, \ldots, \mathbf{F}v_{k-1}, \mathbf{F}v \end{array}}{\mathbf{T}v_k}$$

(23-28) "Last undetermined child" rules for *or* gates, *and* gates and *equiv* gates.

$$\frac{\begin{array}{c} v = \mathrm{even}(v_1, \ldots, v_k) \\ \mathbf{T}v_1, \ldots, \mathbf{T}v_j, j \text{ is even} \\ \mathbf{F}v_{j+1}, \ldots, \mathbf{F}v_{k-1} \\ \mathbf{T}v \end{array}}{\mathbf{F}v_k} \qquad \frac{\begin{array}{c} v = \mathrm{even}(v_1, \ldots, v_k) \\ \mathbf{T}v_1, \ldots, \mathbf{T}v_j, j \text{ is even} \\ \mathbf{F}v_{j+1}, \ldots, \mathbf{F}v_{k-1} \\ \mathbf{F}v \end{array}}{\mathbf{T}v_k} \qquad \frac{\begin{array}{c} v = \mathrm{odd}(v_1, \ldots, v_k) \\ \mathbf{T}v_1, \ldots, \mathbf{T}v_j, j \text{ is odd} \\ \mathbf{F}v_{j+1}, \ldots, \mathbf{F}v_{k-1} \\ \mathbf{T}v \end{array}}{\mathbf{F}v_k}$$

$$\frac{\begin{array}{c} v = \mathrm{even}(v_1, \ldots, v_k) \\ \mathbf{T}v_1, \ldots, \mathbf{T}v_j, j \text{ is odd} \\ \mathbf{F}v_{j+1}, \ldots, \mathbf{F}v_{k-1} \\ \mathbf{T}v \end{array}}{\mathbf{T}v_k} \qquad \frac{\begin{array}{c} v = \mathrm{even}(v_1, \ldots, v_k) \\ \mathbf{T}v_1, \ldots, \mathbf{T}v_j, j \text{ is odd} \\ \mathbf{F}v_{j+1}, \ldots, \mathbf{F}v_{k-1} \\ \mathbf{F}v \end{array}}{\mathbf{F}v_k} \qquad \frac{\begin{array}{c} v = \mathrm{odd}(v_1, \ldots, v_k) \\ \mathbf{T}v_1, \ldots, \mathbf{T}v_j, j \text{ is odd} \\ \mathbf{F}v_{j+1}, \ldots, \mathbf{F}v_{k-1} \\ \mathbf{F}v \end{array}}{\mathbf{T}v_k}$$

$$\frac{\begin{array}{c} v = \mathrm{odd}(v_1, \ldots, v_k) \\ \mathbf{T}v_1, \ldots, \mathbf{T}v_j, j \text{ is even} \\ \mathbf{F}v_{j+1}, \ldots, \mathbf{F}v_{k-1} \\ \mathbf{T}v \end{array}}{\mathbf{T}v_k} \qquad \frac{\begin{array}{c} v = \mathrm{odd}(v_1, \ldots, v_k) \\ \mathbf{T}v_1, \ldots, \mathbf{T}v_j, j \text{ is even} \\ \mathbf{F}v_{j+1}, \ldots, \mathbf{F}v_{k-1} \\ \mathbf{F}v \end{array}}{\mathbf{F}v_k}$$

(29-36) "Last undetermined child" rules for *even* gates and *odd* gates

Figure 5.2: Additional deduction rules for the BC tableau. The ordering of children is as discussed in Figure 5.1.

$$\frac{v = \text{ite}(v_c, v_1, v_2) \quad \mathbf{T}v_c \quad \mathbf{T}v_1}{\mathbf{T}v} \qquad \frac{v = \text{ite}(v_c, v_1, v_2) \quad \mathbf{T}v_c \quad \mathbf{F}v_1}{\mathbf{F}v} \qquad \frac{v = \text{ite}(v_c, v_1, v_2) \quad \mathbf{F}v_c \quad \mathbf{T}v_2}{\mathbf{T}v} \qquad \frac{v = \text{ite}(v_c, v_1, v_2) \quad \mathbf{F}v_c \quad \mathbf{F}v_2}{\mathbf{F}v}$$

$$\frac{v = \text{card}_x^y(v_1, \dots, v_k) \quad \mathbf{T}v_1, \dots, \mathbf{T}v_j,\ j \geq x \quad \mathbf{F}v_{j+1}, \dots, \mathbf{F}v_l,\ k - l \leq y - j}{\mathbf{T}v} \qquad \frac{v = \text{card}_x^y(v_1, \dots, v_k) \quad \mathbf{T}v_1, \dots, \mathbf{T}v_j,\ j < x \quad \mathbf{F}v_{j+1}, \dots, \mathbf{F}v_l,\ k - l < x - j}{\mathbf{F}v}$$

$$\frac{v = \text{card}_x^y(v_1, \dots, v_k) \quad \mathbf{T}v_1, \dots, \mathbf{T}v_j,\ j > y}{\mathbf{F}v}$$

(37-43) “Up”-rules for *ite* and *card* gates.

$$\frac{v = \text{ite}(v_c, v_1, v_2) \quad \mathbf{T}v \quad \mathbf{F}v_1}{\mathbf{T}v_2 \quad \mathbf{F}v_c} \qquad \frac{v = \text{ite}(v_c, v_1, v_2) \quad \mathbf{F}v \quad \mathbf{T}v_1}{\mathbf{F}v_2 \quad \mathbf{F}v_c} \qquad \frac{v = \text{ite}(v_c, v_1, v_2) \quad \mathbf{T}v \quad \mathbf{F}v_2}{\mathbf{T}v_1 \quad \mathbf{T}v_c} \qquad \frac{v = \text{ite}(v_c, v_1, v_2) \quad \mathbf{F}v \quad \mathbf{T}v_2}{\mathbf{F}v_1 \quad \mathbf{T}v_c}$$

$$\frac{v = \text{ite}(v_c, v_1, v_2) \quad \mathbf{F}v_1 \quad \mathbf{F}v_2}{\mathbf{F}v} \qquad \frac{v = \text{ite}(v_c, v_1, v_2) \quad \mathbf{T}v_1 \quad \mathbf{T}v_2}{\mathbf{T}v}$$

(44-49) Additional “Up”-rules and “Down”-rules for *ite* gates

$$\frac{v = \text{card}_x^y(v_1, \dots, v_k) \quad \mathbf{T}v_1, \dots, \mathbf{T}v_j,\ j < x \quad \mathbf{F}v_{j+1}, \dots, \mathbf{F}v_l,\ k - l = x - j \quad \mathbf{T}v}{\mathbf{T}v_{l+1}, \dots, \mathbf{T}v_k} \qquad \frac{v = \text{card}_x^y(v_1, \dots, v_k) \quad \mathbf{T}v_1, \dots, \mathbf{T}v_j,\ j = y \quad \mathbf{T}v}{\mathbf{F}v_{j+1}, \dots, \mathbf{F}v_k}$$

$$\frac{v = \text{card}_x^y(v_1, \dots, v_k) \quad \mathbf{T}v_1, \dots, \mathbf{T}v_j,\ j + 1 = x \quad \mathbf{F}v_{j+1}, \dots, \mathbf{F}v_l,\ j + k - l \leq y \quad \mathbf{F}v}{\mathbf{F}v_{l+1}, \dots, \mathbf{F}v_k} \qquad \frac{v = \text{card}_x^y(v_1, \dots, v_k) \quad \mathbf{T}v_1, \dots, \mathbf{T}v_j,\ j \geq x \quad \mathbf{F}v_{j+1}, \dots, \mathbf{F}v_l,\ j + k - l = y + 1 \quad \mathbf{F}v}{\mathbf{F}v_{l+1}, \dots, \mathbf{F}v_k}$$

(50-53) “Last unassigned children”-rules for $card_x^y$ gates

Figure 5.3: Additional deduction rules for *ite* gates and *card* gates. Note that for *ite* gates, the ordering of children is not arbitrary, but must be identical to that in the defining equation of the gate. For all other gates, the ordering is as discussed in Figure 5.1.

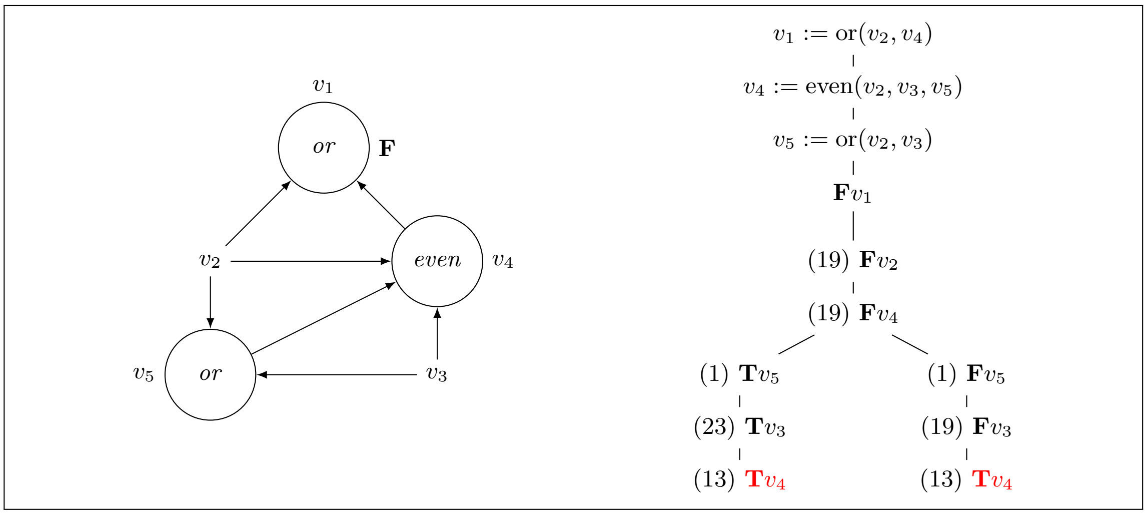


Figure 5.4: An unsatisfiable constrained circuit instance and a corresponding closed BC tableau. Numbers indicate applied tableau rules.

and is true iff at least x, but no more than y inputs are true. In Figure 5.3, we introduce the basic and additional rules used in BattleAx^3 for those gate types.

The BC-tableau proof system is *refutationally sound and complete*, i.e., a constrained circuit is unsatisfiable iff there exists a closed BC tableau for it. For satisfiable instances, the system is also sound and complete, in the sense that if there is an open complete branch, then that branch provides a satisfying truth assignment (soundness) and if the circuit is satisfiable, then any finished tableau contains an open branch (completeness).

5.1.2 Circuit Reduction

In Junttila and Niemelä [26], three strategies are proposed in order to reduce the size of the input circuit in a preprocessing step:

- A *cone-of-influence reduction* is applied. There, instead of solving the original constrained circuit instance (C, τ), an instance (C', τ) is solved, where C' is the subcircuit of C that contains exactly the gates assigned in τ together with all of their descendant gates. Since a consistent assignment for the eliminated gates can be found from a consistent assignment of C' by propagating all assignments for gates in C' to their ancestors via the "Up"-rules presented in Figure 5.1, they can be disregarded in the solving process.

  While we expect this strategy to reduce runtimes on average, they may increase the size of the smallest closed tableau, since an atomic cut on a removed gate (C, τ) essentially simulates a non-analytic cut on (C', τ). Indeed, informal experiments with BattleAx^3 seem to suggest that some instances may be solved faster without this kind of reduction.

$$\frac{v = \text{and}()}{v = \text{true}()} \qquad \frac{v = \text{or}()}{v = \text{false}()} \qquad \frac{v = \text{equiv}()}{v = \text{true}()} \qquad \frac{v = \text{even}()}{v = \text{true}()} \qquad \frac{v = \text{odd}()}{v = \text{false}()}$$

$$\frac{v = \text{and}(v')}{v = v'} \qquad \frac{v = \text{or}(v')}{v = v'} \qquad \frac{v = \text{equiv}(v')}{v = \text{true}()} \qquad \frac{v = \text{even}(v')}{v = \text{not}(v')} \qquad \frac{v = \text{odd}(v')}{v = v'}$$

(i) Simplification rule for 0-ary and 1-ary gates

$$\frac{\begin{array}{c} v = \text{or}(v_1, \ldots, v_{i-1}, v_i, v_{i+1}, \ldots v_k) \\ \mathbf{F}v_i \end{array}}{\begin{array}{c} v = \text{or}(v_1, \ldots, v_{i-1}, v_{i+1}, \ldots v_k) \\ \mathbf{F}v_i \end{array}} \qquad \frac{\begin{array}{c} v = \text{or}(v_1, \ldots, v_{i-1}, v_i, v_{i+1}, \ldots v_k) \\ \mathbf{T}v_i \end{array}}{\begin{array}{c} v = \text{true}() \\ \mathbf{T}v_i \end{array}}$$

$$\frac{\begin{array}{c} v = \text{and}(v_1, \ldots, v_{i-1}, v_i, v_{i+1}, \ldots v_k) \\ \mathbf{T}v_i \end{array}}{\begin{array}{c} v = \text{and}(v_1, \ldots, v_{i-1}, v_{i+1}, \ldots v_k) \\ \mathbf{T}v_i \end{array}} \qquad \frac{\begin{array}{c} v = \text{and}(v_1, \ldots, v_{i-1}, v_i, v_{i+1}, \ldots v_k) \\ \mathbf{F}v_i \end{array}}{\begin{array}{c} v = \text{false}() \\ \mathbf{F}v_i \end{array}}$$

$$\frac{\begin{array}{c} v = \text{even}(v_1, \ldots, v_{i-1}, v_i, v_{i+1}, \ldots v_k) \\ \mathbf{T}v_i \end{array}}{\begin{array}{c} v = \text{odd}(v_1, \ldots, v_{i-1}, v_{i+1}, \ldots v_k) \\ \mathbf{T}v_i \end{array}} \qquad \frac{\begin{array}{c} v = \text{even}(v_1, \ldots, v_{i-1}, v_i, v_{i+1}, \ldots v_k) \\ \mathbf{F}v_i \end{array}}{\begin{array}{c} v = \text{even}(v_1, \ldots, v_{i-1}, v_{i+1}, \ldots v_k) \\ \mathbf{F}v_i \end{array}}$$

$$\frac{\begin{array}{c} v = \text{odd}(v_1, \ldots, v_{i-1}, v_i, v_{i+1}, \ldots v_k) \\ \mathbf{T}v_i \end{array}}{\begin{array}{c} v = \text{even}(v_1, \ldots, v_{i-1}, v_{i+1}, \ldots v_k) \\ \mathbf{T}v_i \end{array}} \qquad \frac{\begin{array}{c} v = \text{odd}(v_1, \ldots, v_{i-1}, v_i, v_{i+1}, \ldots v_k) \\ \mathbf{F}v_i \end{array}}{\begin{array}{c} v = \text{odd}(v_1, \ldots, v_{i-1}, v_{i+1}, \ldots v_k) \\ \mathbf{F}v_i \end{array}}$$

(ii) "Determined child"-simplification rules for *or* gates, *and* gates, *even* gates, and *odd* gates

$$\frac{\begin{array}{c} v = \text{equiv}(v_1, \ldots, v_{i-1}, v_i, v_{i+1}, \ldots, v_k) \\ \mathbf{T}v_i \end{array}}{\begin{array}{c} v = \text{and}(v_1, \ldots, v_{i-1}, v_{i+1}, \ldots, v_k) \\ \mathbf{T}v_i \end{array}} \qquad \frac{\begin{array}{c} v = \text{equiv}(v_1, \ldots, v_n) \\ \mathbf{T}v \\ v_1 \text{ is an input gate} \\ v_2 \text{ is an input or a constant gate} \end{array}}{\begin{array}{c} v = \text{equiv}(v_2, \ldots, v_n) \\ \mathbf{T}v \\ v_1 = v_2 \end{array}}$$

(iii) A "determined child"-simplification rule for *equiv* gates

(iv) "Input gate under true equivalence"-simplification

$$\frac{\begin{array}{c} v = \text{not}(v') \\ v' = \text{not}(v'') \end{array}}{\begin{array}{c} v = v'' \\ v' = \text{not}(v'') \end{array}} \qquad \frac{\begin{array}{c} v = \text{and}(\ldots, v', \ldots, v_1, \ldots) \\ v_1 = \text{not}(v') \end{array}}{\begin{array}{c} v = \text{false}() \\ v_1 = \text{not}(v') \end{array}} \qquad \frac{\begin{array}{c} v = \text{or}(\ldots, v', \ldots, v_1, \ldots) \\ v_1 = \text{not}(v') \end{array}}{\begin{array}{c} v = \text{true}() \\ v_1 = \text{not}(v') \end{array}}$$

(v) Double negation and "$v/\neg v$" simplifications

Figure 5.5: Satisfiability-preserving rewriting rules for BC. A statement $v = v'$ indicates that all occurrences of v are substituted with v' in the circuit.

$$\frac{\begin{array}{c} v := \mathrm{ite}(v_c, v_1, v_2) \\ \mathbf{T}v \end{array}}{v := v_1} \qquad \frac{\begin{array}{c} v := \mathrm{ite}(v_c, v_1, v_2) \\ \mathbf{F}v \end{array}}{v := v_2}$$

$$\frac{\begin{array}{c} v := \mathrm{ite}(v_c, v_1, v_2) \\ \mathbf{T}v_1 \\ \mathbf{T}v_2 \end{array}}{v := \mathrm{true}()} \qquad \frac{\begin{array}{c} v := \mathrm{ite}(v_c, v_1, v_2) \\ \mathbf{F}v_1 \\ \mathbf{F}v_2 \end{array}}{v := \mathrm{false}()}$$

$$\frac{\begin{array}{c} v := \mathrm{card}_x^y(v_1, \ldots, v_{i-1}, v_i, v_{i+1}, v_n) \\ \mathbf{F}v_i \end{array}}{v := \mathrm{card}_x^y(v_1, \ldots, v_{i-1}, v_i, v_{i+1}, v_n)} \qquad \frac{\begin{array}{c} v := \mathrm{card}_x^y(v_1, \ldots, v_{i-1}, v_i, v_{i+1}, v_n) \\ \mathbf{T}v_i \end{array}}{v := \mathrm{card}_{x-1}^{y-1}(v_1, \ldots, v_{i-1}, v_i, v_{i+1}, v_n)}$$

$$\frac{\begin{array}{c} v := \mathrm{card}_x^y(v_1, \ldots, v_k) \\ x \leq 0 \wedge y \geq k \end{array}}{v := \mathrm{true}()} \qquad \frac{\begin{array}{c} v := \mathrm{card}_x^y(v_1, \ldots, v_k) \\ x > y \vee y < 0 \vee x > k \end{array}}{v := \mathrm{false}()}$$

Figure 5.6: Additional rewriting rules for "*ite*" gates and $card_x^y$ gates.

- A structural hashing scheme is employed in order to remove redundant gates. Remember that *expand*(f) refers to the exhaustive replacement of gate-function arguments by their respective associated gate formulas. Given two gates g_1 and g_2 in a constrained circuit (C, τ), if *expand*(f_{g_1}) = *expand*(f_{g_2}), i.e., if they are functionally identical, we can remove g_2 and replace all occurrences of g_2 in the other gate equations with g_1. In structural hashing, some functional equalities are efficiently found by merging structurally identical gates in a bottom-up fashion.

  Such a substitution will yield an equisatisfiable constrained circuit instance as long as neither of the gates involved in merging is assigned a value in τ. Otherwise, it is necessary to propagate the gate assignments in τ in the constrained circuit over the two gates. If, for a merged pair of gates (g_1, g_2), it holds that $\tau(g_1) \neq \tau(g_2)$, then the original constrained circuit is unsatisfiable. If $\tau(g_2)$ is defined, but not $\tau(g_1)$, or vice versa, then we must propagate this assignment to the unassigned gate for the new instance to be equisatisfiable.

- Finally, a number of rewriting rules for creating an equisatisfiable instance are introduced in [26]. These are shown in Figure 5.5. A statement $g_1 = g_2$ in a rewriting rule, where both g_1 and g_2 are gates, indicates that all occurrences of g_1 are replaced by g_2 in all gate equations. Similar to structural hashing, gate assignments must be propagated over gates that are substituted.

  Similar to the case of the tableau rules, we present additional rewriting rules for *ite* gates and *card* gates in Figure 5.6. Although such rules were not presented in the original paper by Junttila and Niemelä [26] where the BC tableau was introduced, we suspect that the same or similar rewriting rules are implemented in BCSat.

The same three strategies were implemented for our solver BattleAx^3. We will give a more detailed description of the reduction process in Section 5.2 and describe a scheme for iterative circuit reduction during the solving process.

5.1.3 Tableau-Based SAT as Generalized DLL

For implementing the BC procedure, Junttila and Niemelä [26] propose to apply the deterministic deduction rules exhaustively before branching with the explicit cut rule. Since the BC-tableau rules extend a branch only with assignments nodes and not with gate equations, we can view a depth-first expansion of the tableau of an instance (C, τ) as a process of iteratively enlarging the partial assignment τ and subsequent backtracking when a branch is closed.

In this way, the BC procedure can be implemented as a generalized DLL-procedure [14] for circuit instances. The application of the deterministic deduction rules is a generalization of BCP, and the atomic cut rule corresponds to a decision. Algorithm 5.1 is such a generalized DLL framework which can be used to structure a tableau-based solver. Our program, BattleAx^3, is a prototypical implementation of this framework, and extends it by many techniques occurring in CNF-based SAT solving.

Algorithm 5.1: Tableau-based SAT procedure in a generalized DLL framework.

```
input  : $(C, \tau)$ - a constrained Boolean circuit
output: Satisfiability of $(C, \tau)$

if applyDeterministicRules($C, \tau$) = Conflict then
    return false;
while true do
    if branchComplete($C, \tau$) then
        return true;
    applyAtomicCut($C, \tau$);
    while applyDeterministicRules($C, \tau$) = Conflict do
        bLevel ← analyze($C, \tau$);
        if bLevel < 0 then
            return false;
        else
            backtrack(bLevel);
```

As Drechsler et al. [14] point out, the link between deduction in the BC tableau and deduction in DLL-based SAT on a Tseitin-transformed circuit instance is very strong. Consider as an example a gate $v = \text{or}(v_1, \ldots, v_n)$ and its translation to CNF via Tseitin, i.e.,

$$(v \vee \neg v_1) \wedge \ldots \wedge (v \vee \neg v_n) \wedge (\neg v \vee v_1 \vee \ldots \vee v_n).$$

Each clause can be considered to encode a number of implications. The BC rules can then be viewed as explicitly representing those implications. The first n binary clauses of form $(v \vee \neg v_i)$, for example, encode both the "Up"-rule stating that v must be true $(v_i \Rightarrow v)$, whenever any input is true, as well as the "Down"-rule that sets all inputs to false if the output v is false $(\neg v \Rightarrow v_i)$. For more complex gate types such as *card* gates, "odd" gates or "even" gates, the CNF translation becomes increasingly cumbersome, since the rules are much more concisely expressible in arithmetic than by enumerating all implications.

At this point, it is interesting to make the following observation regarding the structure-preserving translation procedure presented in [37]. Remember that there, some clauses are possibly omitted from a gate's translation depending on its polarity. Since the clauses correspond to certain rules in the BC tableau, we may build a new tableau system that is still refutationally complete and sound by restricting the rules that are applied to each gate to those that would be encoded in the polarity-based translation.

Consider as an example again the *or* gate shown above. If this gate has positive polarity w. r. t. the formula to be checked, we can restrict the rules that are applied to it to the rules that are encoded in the clause

$$(\neg v \vee v_1 \vee \ldots \vee v_n).$$

To be exact, it is sufficient in this case to restrict rule applications on this gate to the application of the "Last unassigned child" rule, and the **F**-"Up" rule. Such restrictions may be useful in implementation for two reasons. Such a restriction may allow for more efficient implementation of the rules. There may be, on the other hand, drawbacks concerning the increased length of minimal proofs, since we are essentially removing deduction shortcuts from the circuit.

5.1.4 BC with Learning and Non-Chronological Backtracking

A straightforward implementation of the BC procedure would expand branches in a depth-first manner. Upon determining that a branch is closed, such an implementation would continue to expand the most recent branch that is still incomplete. From the framework of Algorithm 5.1, it is easy to see that the BC tableau can be implemented in a way that is very similar to modern implementations of the DLL procedure. It is then also possible to expand a BC-based solver with techniques that are popular in CNF-based SAT such as non-chronological backtracking and learning.

This requires a concept similar to that of the implication graph in CNF-based SAT, only for tableau rules. On a branch $B = (b_1, \ldots, b_{i-1}, b_i, b_{i+1}, \ldots b_k)$, we call a set of tableau nodes $R = \{r_1, \ldots, r_j\} \subseteq \{b_1, \ldots, b_{i-1}\}$ a *reason* for b_i iff we can extend a tableau consisting only of the branch $r_1 \rightarrow \ldots \rightarrow r_j$ to a tableau $r_1 \rightarrow \ldots \rightarrow r_j \rightarrow h_1 \rightarrow \ldots \rightarrow h_l$ by applying a tableau rule, where $b_i \in \{h_1, \ldots, h_l\}$. More informally, a set of tableau nodes R is a reason for another tableau node b iff we can deduce b using only the nodes in R. A reason $R = \{r_1, \ldots, r_j\}$ for a tableau node b is *minimal* iff, for any i with $1 \leq i \leq j$, it holds that $R \backslash \{r_i\}$ is not a reason for b.

Remember that an assignment node is a tableau node of the form $\mathbf{T}v$ or $\mathbf{F}v$. A *reason assignment* A for a node b is the subset of all assignment nodes in a reason R for b. The reason assignment A is *minimal* iff the underlying reason R is minimal.

Note, that there may be multiple minimal reason assignments for a single tableau node. First, a node may be deduced on a branch using different tableau rules or different gates. This is similar to the case of CNF-based SAT, where a single assignment might be derived using multiple clauses. Second, even with a single tableau rule application on a single gate, there may be multiple minimal reasons. When applying the **F** "Up"-rule for an *equiv* gate g, for example, each pair of assignments $(\mathbf{T}v_1, \mathbf{F}v_2)$ where v_1 and v_2 are children of g is a minimal reason assignment for $\mathbf{F}g$.

Remember that a *root tableau* of a constrained circuit instance is its initial tableau before any rule is applied. We can now define the concept of a generalized implication graph in the following way:

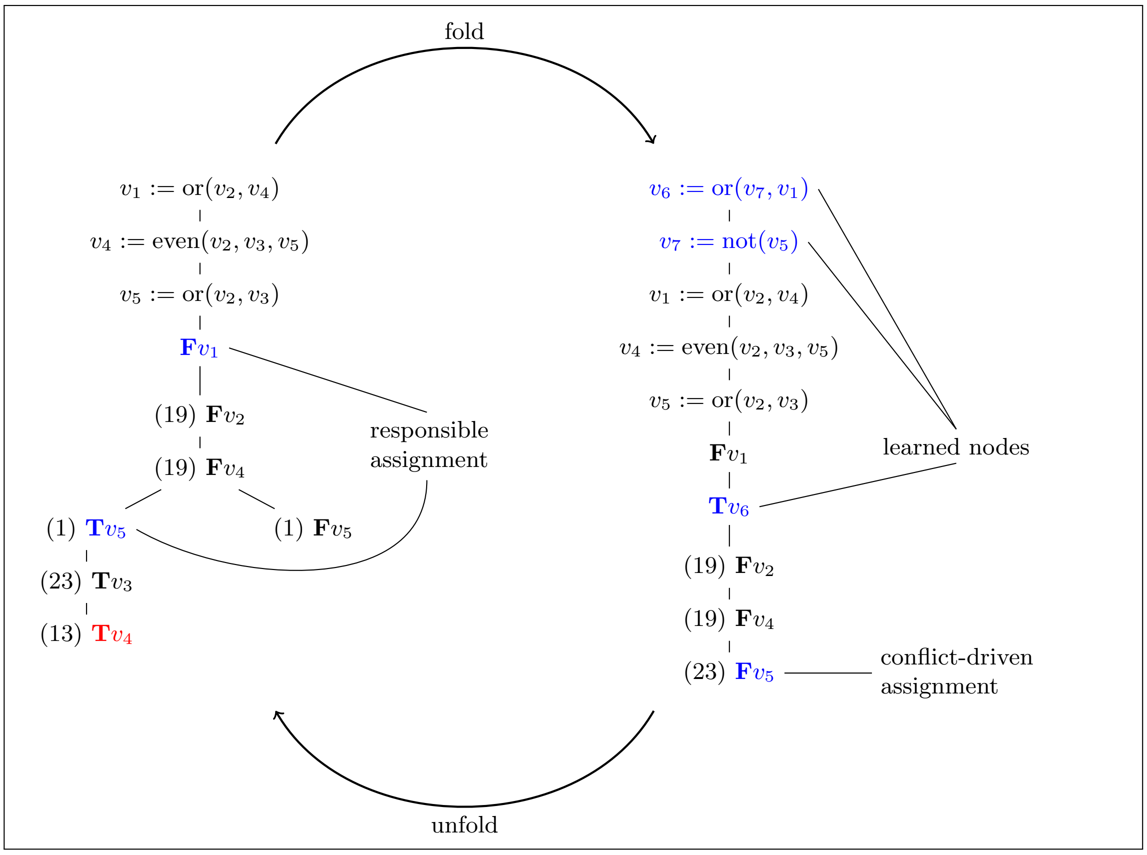
Definition 22. *Let T be a BC tableau for a constrained circuit (C, τ). For a branch $B = (b_1, \ldots, b_k)$ in T, let $\mathrm{Assign}(B)$ denote the set of all assignment nodes in B. We call the the assignment nodes immediately after a cut-rule applications* cut nodes. *Then, a generalized implication graph of B is a graph (V, E) where*

- *$V = \mathrm{Assign}(B) \cup \square$, where the box node $\square$ is a special conflict node,*
- *for every node n in V that is not in the root tableau of (C, τ) and that is not a cut node, the set of antecedents $\{ n' \mid (n', n) \in E\}$ is a minimal reason assignment for n,*
- *for every node n in B that is in the root tableau and for every cut node, the set of antecedents is empty, and*
- *the antecedents of the conflict node $\square$ are the set of tableau nodes $\{ n, n' \mid n = \mathbf{T}v, n' = \mathbf{F}v \} \subset B$.*

We can manage a generalized implication graph in an implementation of a tableau procedure by associating every deduced assignment node n with the assignment nodes that caused n to be deduced. Since there may be multiple minimal reason assignments for a tableau node, multiple implication graphs are possible.

We can use this generalized implication graph analogously to the CNF case in order to perform learning and non-chronological backtracking. Notions such as responsible assignments, separating cuts, and the different strategies for choosing a cut can be extended to this generalized implication graph in a straightforward way. The decision level of a branch is the number of cut rule applications on that branch. A learned clause can be viewed as a fresh *or* gate with its output constrained to true. For the input literals, new *not* gates may have to be added. The resulting structure can then be simply added to the circuit to gain an equisatisfiable instance.

In order to leave the basic BC tableau unchanged, we choose to view learning and non-chronological backtracking as a procedure that transforms a BC tableau for a constrained circuit to a BC tableau for another constrained circuit. The latter constrained circuit is the result of extending the former constrained circuit with the added learned gate and the conflict-driven assertion. This is illustrated in Figure 5.7. We refer to this transformation as *folding* the original tableau with respect to a responsible conflict assignment. The



fold
$v_1 := \mathrm{or}(v_2, v_4)$
$v_4 := \mathrm{even}(v_2, v_3, v_5)$
$v_5 := \mathrm{or}(v_2, v_3)$
$\mathbf{F}v_1$
(19) $\mathbf{F}v_2$
(19) $\mathbf{F}v_4$
(1) $\mathbf{T}v_5$
(1) $\mathbf{F}v_5$
(23) $\mathbf{T}v_3$
(13) $\mathbf{T}v_4$
responsible assignment
$v_6 := \mathrm{or}(v_7, v_1)$
$v_7 := \mathrm{not}(v_5)$
$v_1 := \mathrm{or}(v_2, v_4)$
$v_4 := \mathrm{even}(v_2, v_3, v_5)$
$v_5 := \mathrm{or}(v_2, v_3)$
$\mathbf{F}v_1$
$\mathbf{T}v_6$
(19) $\mathbf{F}v_2$
(19) $\mathbf{F}v_4$
(23) $\mathbf{F}v_5$
learned nodes
conflict-driven assignment
unfold

Figure 5.7: An illustration of learning and non-chronological backtracking in the BC tableau.

reverse process, where all unit-rule applications on learned gates are substituted by subtableaus, we refer to as *unfolding*.

After arriving at a finished folded BC tableau with the generalized DLL procedure, we can exhaustively unfold it in order to obtain a finished tableau of the original constrained circuit instance.

5.1.5 Datastructures for Deduction

Since, similar to CNF-based solvers, the application of deduction rules takes up most of the runtime in a BC-based tableau solver, efficient implementations are of high importance. We have already presented some of these options in Chapter 4. Here, we will give a more compact overview of different implementation techniques for deduction in circuit structures before we explain our implementation choices for BattleAx^3 in the next section. The overview will be based in part on the work in Drechsler et al. [14].

Counter-Based Deduction

A simple possibility for determining the applicability of tableau-rules is to keep counters for each gate that describe the number of positive and negative inputs. Each gate keeps references to its parent gates, and when a formerly unassigned gate is assigned a value, the appropriate counter of the parent is incremented by one. In BCSat, such a counter-based approach was used [25].

Rule applicability conditions can then be easily formulated as algebraic checks on the counters, e.g., if, for an *and* gate g, it holds that the number of children equals the number of true inputs, the *and* "Up"-rule is applicable, and we can deduce $\mathbf{T}g$.

One disadvantage of this approach is that in backtracking, all the counters need to be reset. Compared with stateless or lazy datastructures, this is a significant disadvantage.

Implication Lists

For all those tableau rules in which a single assignment $\mathbf{T}v$ or $\mathbf{F}v$ is a sufficient condition for a rule's applicability, performing algebraic checks on counters is a significant overhead compared with a naive, straightforward implementation. This is similar to the case of binary clauses in CNF, where highly-tuned solvers use direct implication lists. This approach finds applications, for example, in the circuit-solvers presented in Thiffault et al. [46] and Wu et al. [51].

In the BC tableau, we can use implication lists for both the "Up" rule and the "Down" rule for *not* gates, for the **F**-"Up" rule and the **T**-"Down" rule for *and* gates, and for the **F**-"Down" rule and the **T**-"Up" rule for *or* gates.

The use of implication lists changes the structure of the search, since all direct implications are assigned in one step, either before or after tableau rule application. There is disagreement in the literature as to whether this yields performance increases or decreases in the case of CNF-based solvers. In Biere [3], the author argues that the cheaper direct-implications should be used exhaustively before using the more expensive unit-rule. In MiniSAT [39], on the other hand, the direct implication list is directly integrated into the literal watch lists in order to avoid this separation of direct implications and unit-rule applications. The authors argue that the separation of rule applications yields slightly less useful conflict clauses.

Lookup Tables

If a circuit representation is chosen which is restricted to a certain arity for gates, lookup tables can be used in the deduction step. Such an approach is presented in Kuehlmann et al. [29] for an and-inverter-graph (AIG) circuit representation. In an AIG, each gate is either an *and* gate with a fixed arity of two, or a *not* gate.

For a constant-arity gate, each possible combination of true, false and unassigned input and output values can be represented as an entry in a table. These table entries encode the assignments that can be deduced given the current partial assignment. In addition, the table encodes whether a partial input assignment is conflicting. Given a partial assignment to a gate, the table can then be queried in constant time.

Since no state information is saved for gates in this scheme, no action needs to be performed in backtracking to keep rule status consistent.

Generalized Watching Schemes

Some attempts have been made to extend the watching schemes used in CNF-based SAT solvers to gate structures. In Thiffault et al. [46], a watching scheme is proposed for *and* gates and *or* gates.

For *or* gates, such a scheme is virtually identical to the watched-literal scheme in CNF-based SAT. The main difference between clauses and *or* gates concerning deduction is that a clause's output is implicitly constrained to true, while an *or* gate's output may also become false without necessarily causing a conflict.

It is then easy to see how the watched-literal scheme can be translated to circuit structures. For each *or* gate g, two inputs are watched. The invariant over these two watched inputs is that neither of them may become false. If this invariant cannot be made true by moving the watched input after an assignment, then one of two cases holds: If the other watched input is already false, conclude $\mathbf{T}g$. Otherwise, if the other watched input is unassigned, check whether $\mathbf{T}g$ holds. If it does, assign the other watched input to be true. This procedure basically encodes the **F**-"Up" rule and the "Last unassigned child" rule for *or* gates. The other rules are implemented in a straightforward manner.

For *and* gates, the idea is analogous, only with the roles of **T** and **F** values reversed. Two input gates are watched, the invariant being that neither of them may become true. If this invariant cannot be made true by moving watched inputs, the **T**-"Up" rule or the "Last unassigned child rule" may be applicable.

Recently, the solver QuteSAT was presented in Wu et al. [51] in which the watching scheme is extended to arbitrary gate types. The authors distinguish two types of inference over gates, called direct implications and indirect implications. Direct implications are those inference rules that are applicable due to a single assignment. These correspond to binary clauses in the CNF translation of the circuit. Deduction for direct implications is handled via a form of implications lists. Indirect implications are those inference rules whose applicability depends on more than one assignment. For indirect implication, the generalized watching scheme is used.

For the simple gate types *and* and *or*, the watching scheme is similar to the watched-literal scheme on the Tseitin transformed CNF. So, for a gate $v := \mathrm{or}(v_1, \ldots, v_n)$, two of the pairs in $\{(v, \mathbf{F}), (v_1, \mathbf{T}), \ldots, (v_n, \mathbf{T})\}$ would be watched, the invariant being that the variable in a watched pair (v, X) may not be assigned the opposite value to X (this characterization of the scheme is taken from Drechsler et al. [14]). This is similar to simply watching the clause $\neg v \vee v_1 \vee \ldots \vee v_n$ in the watched-literal scheme. For *and* gates, deduction is handled analogously, but with the roles of **T** and **F** reversed.

For more complex gate types, another watching scheme is used. In a gate $v = f(v_1, \ldots, v_n)$, we call a variable in $\{v, v_1, \ldots, v_n\}$ a *watch candidate* iff its assignment can lead to an indirect implication. In such a gate, we watch a number of l watch candidates, the invariant being that the watched variables are *unassigned.*

If the invariant cannot be made true by moving the watch pointer to an as-of-yet unwatched watch candidate, the gate function is used to determine values for the unassigned gates. The number of necessary watched inputs l can be determined by finding the smallest number of assignments k on a gate's inputs and output that trigger an indirect implication. For a gate with m watch candidates, the number of necessary watched inputs is then $l = m - k + 1$.

As an example, consider a gate $v := \text{odd}(v_1, \dots, v_n)$. The set of watch candidates is $\{v, v_1, \dots, v_n\}$, therefore $m = n + 1$. Since we can only make an implied assignment once all but one watch candidates are assigned, it holds that $k = n$ and $l = m - k + 1 = (n + 1) - n + 1 = 2$. We therefore need to watch two candidates. Once it is impossible to uphold the invariant, i.e., when all but one watch candidates are assigned, we use the *odd* gate-function in order to determine the remaining assignment.

Note that, for some gates, the value of l may be rather high. For an *equiv* gate of arity n, for example, $l = n$. In such cases, the use of watch pointers may become more inefficient than using counters.

5.2 Implementing an Enhanced Tableau-Based Solver

While the basic DLL algorithm is conceptually rather simple, modern SAT solvers are highly complex pieces of software whose components interact non-trivially. Implementing a SAT solver calls for many small implementation choices which are rarely described in detail in the literature. Seemingly small changes in the implementation can cause big effects. Biere [3] describes, for example, how an undocumented detail of the implementation of the learning strategy in Chaff [34] improved performance considerably.

Some of these omissions may be rooted in an underestimation of the importance of a given implementation choice. More often though, these omissions can be attributed to a lack of space. An exhaustive description of a solver implementation is simply beyond the scope of most papers.

In this section, we will describe a prototypical implementation of the framework presented in Section 5.1. We have developed BattleAx^3, a BC-based SAT solver with non-chronological backtracking and learning. Furthermore, the basic BattleAx^3 implementation was extended with a number of techniques from CNF-based SAT in order to evaluate their efficiency in circuit instances.

Since we do not suffer from the space constraints of a paper, we have chosen to give a fairly detailed description of BattleAx^3, although not an exhaustive one. We give some information about implementation basics, which are otherwise rather sparse in the literature. Besides studying existing implementations of CNF-based SAT solvers whose source code is openly available (e.g., MiniSAT [39]), the descriptions in Biere [4] proved very helpful for solving many technical details.

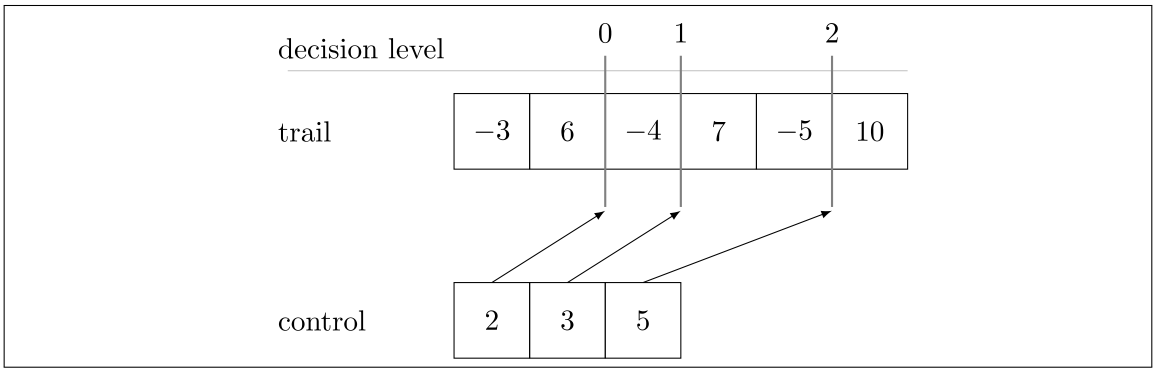


Figure 5.8: Trail stack and control stack.

5.2.1 Basic Datastructures

In BattleAx^3, the circuit is represented as an array of gates. Each gate structure contains a field indicating the type of the gate, a reference to an array containing the array indices of children, and a reference to a second array indicating the array indices of parents.

The current partial assignment is also stored in an array, which indicates for each gate index whether the gate is unassigned, true, or false. In order to keep track of the order of assignments and of the assignments made on each decision level, we use two stacks.

One stack, referred to as the *trail stack*, contains all assignments that were made on the current branch of the tableau in the order in which they were made. Essentially, it is a direct representation of the assignment nodes on the current tableau branch. An assignment node is represented as an integer indicating the index of the assigned array. A positive integer denotes an assignment node of the form $\mathbf{T}v$, a negative integer denotes an assignment node of form $\mathbf{F}v$. Whenever a variable is assigned a value, either due to a decision or a deduction, it is put on top of this stack.

The second stack, called the *control stack*, contains values indicating the size of the trail stack at the end of each decision level. Initially, the control stack is empty. Each time before a decision is made, i.e., before the decision level is increased, the current size of the trail stack is put on the control stack. In backtracking, the information on the control stack is used to identify the decisions that have to be undone. An illustration of both stacks is given in Figure 5.8.

5.2.2 Deduction

The main datastructure involved in the deduction process is the *assignment queue.* Whenever a variable is assigned, it is appended to the assignment queue. In the deduction step, each assignment in turn is taken off the queue and the necessary deduction rules are applied. These may generate new assignments, which are again appended to the queue. The whole process is repeated until either a conflicting assignment is produced or until the assignment queue is empty. The procedure is sketched in Algorithm

5.2.

Algorithm 5.2: The applyDeterministicRules step in Algorithm 5.1.

```
while ¬isEmpty(assignmentQueue) do
    assignment ← removeFirst(assignmentQueue);
    deducedAssignments ← fireApplicableRules(assignment);
    $\tau \leftarrow \tau \cup$ deducedAssignments;
    appendAll(assignmentQueue, deducedAssignments);
```

The fireApplicableRules step in Algorithm 5.2, can be implemented for an assignment v as follows.

(i) Check for all parents of var(v) whether any "Up" rules or "Last unassigned child" rules are applicable.

(ii) Check for gate v itself whether any "Down" rules or "Last unassigned child" rules are applicable

Instead of a queue, a stack could also be used to store assignments. This would lead to a depth-first exploration of deduced assignments, in contrast to the breadth-first exploration effected by the queue implementation we use. Biere [3] remarks that the use of a queue heuristically minimizes the implication graph, which keeps the length of learned clauses low.

Biere [3] also explores the use of *lazy assignments*. This means that whenever a variable assignment is deduced or decided, it is only put on the assignment queue. The actual variable assignment is delayed until the variable is dequeued during the deduce step. This contrasts with the more common technique of *busy assignments*, i.e., setting its value and putting it on the trail stack as soon as its value is deduced or decided. The advantage in lazy assignments is that, since a variable assignment is not made immediately after it is deduced, it may be deduced a second time by another clause. In this way, multiple reasons can be recorded for each variable assignment.

Biere [3] suggests that such a strategy is not ideal. While lazy assignments allow for the identification of multiple reasons for a single conflict which can be heuristically used to produce small conflict clauses, it leads to complications in the overall design of a solver which are not worth the possible speedups. In our implementation, we therefore use busy assignments, which is the implementation choice in most solvers.

We have chosen to implement deduction using three distinct implementation strategies. For normal gates, counters are used. While this may not be the most efficient choice on average, it is easy to implement and allows for equally efficient encoding of all gate rules in the BC tableau. The BC tableau includes not only simple gate types, but complex ones such as *equiv* and *card* gates. A generalized watched-input scheme as in Wu et al. [51] would likely be inefficient on instances that rely heavily on those gate types since the rules would need to be checked nearly as often as in a counter-based approach, but each check requires effort linear in the size of the gate. It is still likely that the average runtime of the solver could be decreased for many classes of instances if a watched pointer scheme was implemented for some gate types.

As a second type of deduction, we have implemented the option to use implication lists for simple rules whose firing condition depends on a single variable assignment. If this option is activated, all implication-list deductions are made before the ordinary tableau rules are fired. This changes the structure of the search since deductions are made in a different order which leads to a different implication graph. Furthermore, there is an option to apply implication-list deduction *exhaustively* before going on to the application of the remaining rules, as it is suggested in Biere [3].

The third kind of deduction is used for learned clauses. While it would be possible to represent learned clauses as *or* gates with outputs that are constrained to true, a direct representation seems more natural and is likely to be more efficient. We use the watched-literal scheme for deduction [34]. The watched literals are kept at the first and second position of a clause as it is outlined in Biere [5], which avoids the use of a separate watcher datastructure.

5.2.3 Conflict Analysis, Learning and Backjumping

We have already introduced the notion of a generalized implication graph for the BC tableau procedure. Our solver successively builds up such a graph for the current branch by storing a minimal reason assignment for each derived tableau node. In CNF-based SAT, the implication graph is usually represented implicitly by storing only a pointer to a clause for each assignment. The full graph can be retrieved by analyzing the literals in these clauses. Since the tableau format is more heterogeneous than CNF, we explicitly associate a stack of integers with each gate which record its reason assignments.

In some cases, multiple reason assignments may be found for a single tableau rule application. While it would be interesting to evaluate different strategies for choosing among those reasons, we use no such heuristic, but simply use the first applicable reason assignment that is identified in the analysis step.

In Kuehlmann et al. [29], an alternative way of implementing an implication graph is presented. There, each circuit gate g is associated with a bit vector representing the decisions that are ultimately responsible for the current value of g. If two gate values are used to deduce a third, the third value is associated with the result of applying a bit-wise *or* operation to the bit vectors of the first two gates. When a conflict is derived, a bit-wise *or* operation on the conflicting gate assignments' bit vectors instantly yields a responsible assignment. This strategy can be seen as a busy variant of the analysis procedure in modern CNF-based SAT solvers. When a conflict is derived, we immediately gain a responsible assignment, but more work needs to be done during deduction. Such an approach is not ideal for our purposes since it would entail a large amount of overhead during deduction in large circuits. Furthermore, it is not clear how one can implement efficient learning approaches such as the first-cut scheme using this technique.

We use the first-cut scheme for clause learning that was already presented in Chapter 3. In the first-cut scheme, the conflict clause contains the UIP closest to the conflict node (the *first UIP*) together with all assignment nodes from earlier decision levels that have an edge to an assignment on the conflict side of the partition induced by the UIP.

The identification of the first UIP can be performed in a single traversal of the implication graph by going backwards from the conflict node and counting "open paths". Algorithm 5.3 (taken from [4]) sketches an efficient procedure for identifying the conflict clause induced by the first cut. The number of open paths is initialized to the antecedents of the conflict node in the generalized implication graph. Then the implication graph is traversed backwards from the conflict node in the order indicated by the trail stack (i.e., the most recent assignment node is visited first), ignoring all assignments nodes that do not have a path to the conflict node. When a node is visited, the number of open paths is decremented by one, and incremented by the number of unvisited children that are on the most-recent decision level. When the number of open paths reaches one, the next assignment node that is visited is the UIP.

Algorithm 5.3: Identifying the first cut.

```
conflictClause ← $\emptyset$;
openPaths ← 0;
for each assignment ∈ conflictReason do
    if getDLevel(assignment) = conflictDLevel then
        mark(assignment);
        openPaths ← openPaths + 1;
    else
        conflictClause ← conflictClause ∪ { litMadeFalseBy(assignment) };
while true do
    assignment ← pop(trailStack);
    if isMarked(assignment) then
        if openPaths = 1 then
            return conflictClause ∪ { litMadeFalseBy(assignment) };
        for each r ∈ reason(assignment) do
            if getDLevel(r) = conflictDLevel then
                mark(r);
                openPaths ← openPaths + 1;
            else
                conflictClause ← conflictClause ∪ { litMadeFalseBy(r) };
        openPaths ← openPaths − 1;
```

Learned clauses are stored in a monolithic memory array. This necessitates additional garbage collection code, but is slightly more cache efficient than using normal memory allocation. Clauses are deleted based on their recent activity in a VSIDS-like scheme. Each clause is associated with a counter. This counter is increased twice each time a clause is conflicting, and once for each clause that was used to derive the conflict on the most recent decision level.

5.2.4 Circuit Reduction

As was explained earlier, three techniques for reducing the size of a circuit are implemented,

- a cone-of-influence reduction,
- a structural hashing scheme, and
- the rewriting rules in Figure 5.5 and Figure 5.6.

These rules are applied at preprocessing and whenever a one-literal clause is learned. A one-literal clause triggers a backtrack to decision level 0 and effectively restarts the solver with an added constraint on the circuit instance. This added constraint allows a reapplication of the rewriting rules, which in turn opens up new possibilities for reduction with the other two approaches. In some instances, one-literal clauses are learned very frequently. There, the resulting frequent circuit reductions can take up a significant amount of the runtime. In order not to impede overall efficiency in such cases, we do not apply circuit reduction after every learned one-literal clause, but count the number of learned one-literal clauses since the last reduction step. When this number reaches a fixed threshold, circuit reduction is applied.

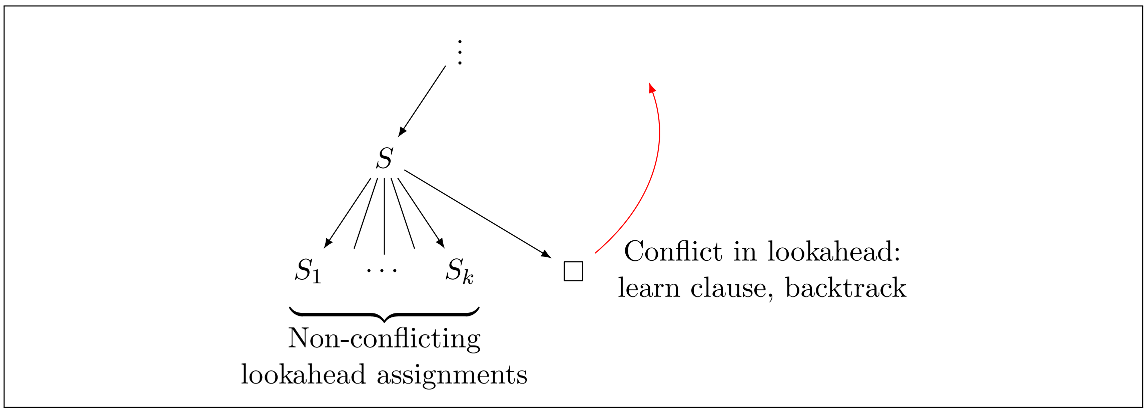
After reducing the circuit, the following rules are applied to each clause c in the clause database:

- If a literal in c corresponds to a gate that was removed in the cone-of-influence reduction, remove c from the clause database.
- If a literal l in c corresponds to gate that was substituted by another gate using the rewriting rules or the structural hashing scheme, replace l by the same-phase literal on the substitute gate.
- If a literal l in c is false at decision level 0, remove l from c.
- If a literal l in c is true at decision level 0, remove c from the clause database.

5.2.5 One-Step Lookahead

We have implemented the option of using a one-step lookahead rule, which was proposed in Junttila and Niemelä [26]. This rule states that, if we can deduce a conflict using the deterministic deduction rules after "trying out" a lookahead assignment $\mathbf{T}v$ ($\mathbf{F}v$) on a branch, we can immediately deduce $\mathbf{F}v$ (resp. $\mathbf{T}v$).

In a solver with non-chronological backtracking and learning, we can implement lookahead as a straightforward decision. The difference is that if no conflict occurs as an immediate result of the decision, we backtrack chronologically and possibly try out a number of other lookahead assignments in the same way. If, on the other hand, a conflict occurs as a result of the lookahead assignment, we can learn a conflict clause using the first-cut scheme and backtrack non-chronologically. Considering the lookahead rule from the perspective of the generalized DLL procedure, this is essentially an introduction of local breadth-first search at each node in the depth-first search tree. This idea is sketched in Figure 5.9.



S
S_1
$\cdots$
S_k
$\square$
Conflict in lookahead: learn clause, backtrack
Non-conflicting lookahead assignments

Figure 5.9: A partial search tree of the generalized DLL procedure using the lookahead rule.

In Junttila and Niemelä [26], the authors seem to propose an unrestricted application of the lookahead heuristic where, in each step of the search, all unassigned variables are used as lookahead assignments. Such an implementation was evaluated in BattleAx^3, but was found to be highly inefficient on some benchmarks. We have implemented a number of different restrictions in order to try increasing the robustness of the strategy. One possible restriction is to apply lookahead only to children of unjustified gates. Another one is to combine the lookahead scheme with the VSIDS heuristic for variable selection.

In our implementation, VSIDS literals are stored in a heap so that the literal with the highest score can be retrieved efficiently during the decision step. The heap itself is implemented as an array. We can restrict our lookahead to the first n variables found in the heap's literal array. While these may not be the n highest-scoring VSIDS literals, it still yields an efficiently computable heuristic for applying lookahead only for variables that are somewhat likely to lead to conflicts.

The implementation of the lookahead rule can be sped up significantly by marking assignments that are derived during a lookahead. Such assignments cannot lead to conflicts and therefore do not need to be tried out as lookahead assignments on the same decision level. As an example, consider a lookahead assignment $\mathbf{T}v_l$ from which $\mathbf{F}v$ can be derived inside the lookahead. Assume that the lookahead assignment does not lead to a conflict. Now it is clear that in subsequent lookaheads on the same decision level, we can skip the lookahead assignment $\mathbf{F}v$, since this assignment was already made as a consequence of $\mathbf{T}v_l$.

5.2.6 Decision Variable Selection

We have implemented three different strategies for choosing decision variables: the popular VSIDS and BerkMin heuristics, both of which were described in Chapter 3, and the *lookahead decision heuristic* that builds upon the lookahead rule described in the previous section and was the decision heuristic used in BattleAx^3.

The VSIDS scheme is implemented using a priority queue with an underlying heap. During decision variable selection, elements are repeatedly removed from the front of the queue until a literal is found whose variable is unassigned. In BerkMin, the decision heuristic chooses a literal from the the top clause, which is the most-recently learned clause that is not satisfied. A naive implementation can lead to a majority of the runtime being spent in the decision heuristic in search of the top clause. This can be solved by caching its position between conflicts.

In the lookahead heuristic, we count the number of rule applications that can be performed as a consequence of the lookahead assignments. Let $v^{\mathbf{F}}$ and $v^{\mathbf{T}}$ be the number of new assignment nodes that can be deduces from an assignment $\mathbf{F}v$ respectively $\mathbf{T}v$. The lookahead heuristic then chooses a decision variable v for which $\min(v^{\mathbf{F}}, v^{\mathbf{T}})$ is maximal.

In addition to the heuristics outlined above, each heuristic can be combined with justification filtering and phase saving. In justification filtering, only those variables are considered for decisions which have unjustified parents. Justification filtering entails significant overhead since a justification frontier has to be maintained.

Phase saving was already described in Chapter 3. Here, the phase of a decision variable v is set to be the last phase that variable was assigned to. In this way, some of the progress that was lost due to non-chronological backtracks and restarts can quickly be regained.

5.2.7 Restarts

For our solver BattleAx^3, we also adopt the use of random restarts. Here, the solver is reset at certain points during the search to decision level 0, but with the database of learned clause intact. We introduce some transient randomness to allow the solver to enter new regions of the search space.

Restarts are usually triggered after a certain number of conflicts or decisions. This number should be successively increased in order for the solver to stay complete. Choosing such a sequence of restart limits is a main consideration in implementing restarts.

We have implemented three different strategies for this purpose:

- A simple strategy where the restarting and clause limit are increased geometrically, starting with an initial value of 100 for the restarting limit, and the number of gates divided by two for the clause limit. At each restart, the restarting limit is increased by 50% and the clause limit by 10% (parameters taken from [39]).

- The nested restarting strategy used in PicoSAT [5], where two limits are used, the actual restart limit and an outer limit (both with an initial value of 100). The restart limit is increased by 10% at each restart. Upon reaching the outer limit, the restart limit is reset to an initial value, and the outer limit is increased by 10%. The clause limit is increased by 5% each time the outer limit is increased.

- The RSAT2.0 strategy [36] (originally motivated by the work in Huang [22]), where the Luby sequence [31] is used as a basis for the restart limit sequence. The Luby sequence is the sequence $t_1, t_2, t_3, \ldots$ such that

  $$t_i = \begin{cases} 2^{k-1}, & \text{if } i = 2^k - 1 \\ t_{i-2^{k-1}+1}, & \text{if } 2^{k-1} \leq i < 2^k - 1 \end{cases}$$

  and has the prefix

  $$1, 1, 2, 1, 1, 2, 4, 1, 1, 2, 1, 1, 2, 4, 8, 1, 1, 2, 1, 1, 2, 4, 1, 1, 2, 1, 1, \ldots$$

  We follow the suggestion in Pipatsrisawat and Darwiche [36] and set the unit for the Luby sequence to 512 conflicts, i.e., the actual sequence of restart limits is $512, 512, 1024, 512, \ldots$. Furthermore, we increase the clause limit by 15% every 2^k restarts.

At each restart, we perform circuit reduction if any one-literal clauses were learned during the last restart interval.

Chapter 6

Results

In this chapter, we will evaluate some of the techniques implemented in BattleAx^3 using a benchmark set of 38 instances introduced in Järvisalo and Niemelä [23]. The benchmarks are given in the BC1.0 format [24] which allows to express circuit structures by defining their gate equations. The problems are taken from the domains of

- verification of superscalar processors,
- integer factorization based on hardware multiplier designs,
- equivalence checking of hardware multipliers,
- bounded model checking for deadlocks in LTSs, and
- linear temporal logic BMC of finite state systems.

Some simple statistics over the instances in the benchmark set, including the number of gates, the average number of children in non-input gates, and the average and maximum path lengths between primary inputs and outputs, are shown in Table 6.1. The numbers are derived from BattleAx^3's parsing engine. Note that, due to some implementation choices, the number of gates shown here may be slightly higher than the actual number of gates defined in the instances.

All benchmarks presented in this chapter were performed on a Linux workstation with 2 GB of RAM and an Intel Core 2 Quad Q6600 processor overclocked to 3.0 GHz. In each case, two benchmark instances were run concurrently, as this did not seem to affect overall runtime. The timeout was set at 1800 seconds.

6.1 Comparing Decision Heuristics

In Table 6.2, a comparison between the VSIDS and BerkMin heuristic is given. In VSIDS, the literal with maximum associated score is chosen for assignment, while in BerkMin, the maximum-score literal of the top clause, i.e., the most recently learned clause that is not satisfied, is chosen. According to Goldberg and Novikov [19], the

#	**instance name**	**no. gates**	**avg. no. children**	**avg. path (in→out)**	**max. path (in→out)**
1	1394-4-3.p1neg.k10.unsat	67904	2.02	9.86	212
2	1394-4-3.p1neg.k11.sat	74357	2.02	9.86	213
3	1394-5-2.p0neg.k13.unsat	88882	1.87	9.46	194
4	atree.sat.34.0	12965	2.48	4.90	63
5	atree.sat.36.50	14333	2.49	4.98	66
6	atree.sat.38.100	15469	2.49	5.11	66
7	atree.unsat.32.0	11461	2.48	5.07	50
8	atree.unsat.34.50	12965	2.48	4.90	63
9	atree.unsat.36.100	14333	2.49	4.98	66
10	braun.sat.32.0	6176	1.99	7.89	127
11	braun.sat.34.50	6970	1.99	7.90	135
12	braun.sat.36.100	7812	1.99	7.94	143
13	braun.unsat.32.0	6176	1.99	7.89	127
14	braun.unsat.34.50	6970	1.99	7.90	135
15	braun.unsat.36.100	7812	1.99	7.94	143
16	brp.ptimonegnv.k23.unsat	18087	1.82	8.11	214
17	brp.ptimonegnv.k24.sat	18869	1.82	8.11	223
18	csmacd.p0.k16.unsat	105550	2.23	8.43	147
19	dme3.ptimo.k61.unsat	57973	1.92	7.60	1052
20	dme3.ptimo.k62.sat	58921	1.92	7.60	1069
21	dme3.ptimonegnv.k58.unsat	55020	1.93	7.80	1002
22	dme3.ptimonegnv.k59.sat	55966	1.93	7.80	1019
23	dme5.ptimo.k65.unsat	100221	1.94	7.62	1120
24	dp_12.i.k10.unsat	5907	1.82	3.23	61
25	eq-test.atree.braun.10.unsat	1978	2.27	3.51	41
26	eq-test.atree.braun.8.unsat	1320	2.26	3.45	33
27	eq-test.atree.braun.9.unsat	1745	2.28	3.58	37
28	fvp-unsat.2.0.3pipe.1	2879	10.15	6.98	42
29	fvp-unsat.2.0.3pipe_2_ooo.1	8299	5.68	6.83	51
30	fvp-unsat.2.0.4pipe_1_ooo.1	19640	6.14	6.29	77
31	fvp-unsat.2.0.4pipe_2_ooo.1	19966	6.44	6.91	77
32	fvp-unsat.2.0.5pipe_1_ooo.1	37648	7.45	6.33	123
33	key_4.p.k28.unsat	21765	1.89	3.41	198
34	key_4.p.k37.sat	32727	1.96	3.70	261
35	key_5.p.k29.unsat	27354	1.90	3.37	205
36	key_5.p.k37.sat	39210	1.96	3.63	261
37	mmgt_4.i.k15.unsat	10575	2.10	3.22	107
38	q_1.i.k18.unsat	11064	1.76	2.66	123

Table 6.1: The BC benchmark set from Järvisalo and Niemelä [23].

BerkMin strategy makes the decision heuristic more robust by increasing the speed with which the solver adjust to new regions of the search space.

Overall, both heuristics seem to work nearly equally well in BattleAx^3. The BerkMin heuristic proves fast for small instances, and manages to solve one more instance than VSIDS in the 30-minute time limit. Still, VSIDS seems to be the better overall choice with a lower combined runtime and a smaller number of average decisions.

An interesting observation can be made when studying the average clause lengths and average decision levels of both strategies. Here, values differ drastically and in a rather counter-intuitive way. VSIDS produces very long conflict clauses, but does not, on average, spend much of the search deep down in the search tree. BerkMin's average decision level is three times deeper than that of VSIDS and the conflict clauses are only half as long on average, but it is slower overall.

First of all, this is an indication that the length of short conflict clauses is not a very good indicator of the efficiency of a decision heuristic. Second, it may be an indication that increasing the locality of the search, as is done in BerkMin, may delay conflicts that would otherwise occur earlier, and therefore increase the average decision level.

The vast differences in average clause length and average decision level are rather unexpected considering that both the BerkMin decision heuristic and VSIDS are based on similar ideas. It might be worthwhile to analyze the interplay between decision heuristics, average decision level, and average conflict complexity more closely. A better understanding of these effects may yield, for example, more efficient decision heuristics or meta-heuristics which dynamically alter decision heuristic parameters during the search.

6.2 Evaluating Lookahead

We have implemented the lookahead rule and the lookahead decision heuristic from Junttila and Niemelä [26], both of which are described in detail in Chapter 5. We have tried to answer the question of whether the application of this rule, which seems to have worked well in BCSat [26], still manages to improve solving efficiency on the extended tableau procedure.

In Table 6.3, a basic comparison between a straightforward application of the lookahead rule and lookahead decision heuristic, a restricted application of the same two techniques, and the normal VSIDS-based approach are presented. It is obvious that the unrestricted application of the lookahead rule decreases solving efficiency considerably. At the same time, the number of actual decisions on the solved instances is decreased by two orders of magnitude using this strategy. Restricting the lookahead to the children of unjustified gates seems to increase efficiency somewhat compared to an unrestricted lookahead, but there is still a significant performance drop compared to the VSIDS baseline.

In Table 6.4, we have tried to explore restrictions of the basic lookahead rule further. Again, the left column shows the baseline results for the VSIDS decision heuristic without lookahead. We compare this with a restriction of the lookahead rule to the first 4 and 32 literals in the VSIDS heap array (4VSIDSLH and 32VSIDSLH). We also evaluate using

		VSIDS				**BerkMin**			
#	**sat**	**time (s)**	**dec-isions**	**clause length**	**avg. dLevel**	**time (s)**	**dec-isions**	**clause length**	**avg. dLevel**
1	no	**139.99**	280253	75.76	74.06	289.63	730389	65.96	86.10
2	yes	**104.00**	397522	47.55	115.40	292.34	917344	28.66	86.94
3	no	483.01	522928	157.54	86.07	**448.20**	759067	66.08	74.20
4	yes	**132.80**	189790	284.93	19.59	259.91	398720	255.78	15.86
5	yes	**250.61**	318747	239.52	28.12	339.70	561101	193.93	19.63
6	yes	**83.23**	137902	139.57	25.67	595.02	711945	285.35	20.38
7	no	381.01	484422	577.70	20.49	**343.83**	430269	234.84	13.13
8	no	**403.24**	425575	552.88	16.69	911.96	962775	596.54	13.89
9	no	> 1800				**1583.92**	1402523	441.29	14.36
10	yes	**46.08**	132492	156.57	17.63	132.54	345499	149.20	11.73
11	yes	460.68	721780	354.73	22.61	**354.18**	711594	185.29	13.49
12	yes	336.37	722126	376.44	25.37	**8.93**	55827	119.14	18.01
13	no	165.92	323037	381.88	20.57	**110.45**	309481	125.01	11.78
14	no	**246.14**	632974	298.01	20.02	290.53	708569	169.11	11.54
15	no	809.52	1215931	323.65	27.50	**788.31**	1629242	219.75	12.53
16	no	**121.60**	224553	111.83	25.65	124.11	262040	75.40	18.09
17	yes	**57.86**	130666	77.60	34.39	102.66	226141	81.61	20.06
18	no	**1024.70**	639245	163.00	52.18	> 1800			
19	no	**211.44**	506279	322.71	78.02	262.61	705258	219.81	498.50
20	yes	410.38	707957	389.89	157.39	**30.01**	660122	63.97	1634.63
21	no	69.14	242506	207.22	79.72	**59.16**	512893	111.23	677.01
22	yes	44.11	222240	178.68	411.80	**12.99**	295024	33.34	1108.36
23	no	**47.40**	238891	146.98	103.62	1657.66	1454497	266.10	604.01
24	no	> 1800				> 1800			
25	no	**820.30**	4827722	103.20	13.65	1251.49	6559621	82.39	12.42
26	no	**14.30**	164324	67.04	11.66	21.08	256897	57.64	9.98
27	no	122.89	993259	80.38	12.16	**118.34**	1063909	62.08	11.07
28	no	1.72	59901	83.08	47.45	**1.71**	55296	93.12	35.19
29	no	2.56	48552	106.52	34.68	**1.26**	23051	82.22	18.54
30	no	11.02	129459	186.85	43.78	**7.76**	67413	181.59	21.33
31	no	21.12	187714	237.68	47.77	**10.81**	80290	187.46	21.29
32	no	41.78	296308	310.34	77.55	**30.66**	140095	323.45	24.85
33	no	139.09	198997	692.28	62.02	**138.22**	287455	315.04	17.03
34	yes	37.69	95608	399.16	66.64	**22.55**	140564	123.67	32.89
35	no	363.86	354655	993.65	68.84	**194.68**	287101	420.37	21.26
36	yes	464.60	531818	854.91	96.58	**2.90**	37924	29.39	39.93
37	no	1051.50	1703756	410.92	45.21	**638.07**	1155698	242.71	15.70
38	no	> 1800				**460.53**	895742	274.21	14.52
		lower bound	**avg. over solved**			**lower bound**	**avg. over solved**		
		> **14521.66**	543139.69	288.31	59.73	> 15498.71	737182.17	184.65	150.87
		inst.solved by both:				**inst.solved by both:**			
		sum	**avg.**			**sum**	**avg.**		
		8096.96	540313.05	291.99	59.96	9854.26	691267	169.04	154.46
# solved		35				**36**			

Table 6.2: Comparing VSIDS and BerkMin decision heuristics.

#	**sat**	**VSIDS**		**LH dec. heur.**		**LH dec. heur.**	
		(no lookahead)		**(unrestricted)**		**(just. restr.)**	
1	no	**139.99**	280253	> 1800		> 1800	
2	yes	**104.00**	397522	> 1800		> 1800	
3	no	**483.01**	522928	> 1800		> 1800	
4	yes	**132.80**	189790	> 1800		> 1800	
5	yes	**250.61**	318747	> 1800		1135.73	32677
6	yes	**83.23**	137902	> 1800		> 1800	
7	no	**381.01**	484422	> 1800		1637.16	27571
8	no	**403.24**	425575	> 1800		> 1800	
9	no	> 1800		> 1800		> 1800	
10	yes	**46.08**	132492	809.23	6977	796.52	38687
11	yes	**460.68**	721780	1343.05	9156	675.37	40267
12	yes	**336.37**	722126	> 1800		563.59	30065
13	no	**165.92**	323037	736.22	4907	645.30	57473
14	no	**246.14**	632974	> 1800		1677.35	62624
15	no	**809.52**	1215931	> 1800		> 1800	
16	no	**121.60**	224553	> 1800		> 1800	
17	yes	**57.86**	130666	> 1800		> 1800	
18	no	**1024.70**	639245	> 1800		> 1800	
19	no	**211.44**	506279	> 1800		> 1800	
20	yes	**410.38**	707957	> 1800		> 1800	
21	no	**69.14**	242506	> 1800		> 1800	
22	yes	**44.11**	222240	> 1800		> 1800	
23	no	**47.40**	238891	> 1800		> 1800	
24	no	> 1800		> 1800		> 1800	
25	no	**820.30**	4827722	> 1800		1277.79	459253
26	no	**14.30**	164324	119.67	7899	30.71	33837
27	no	**122.89**	993259	1738.33	50102	190.82	105418
28	no	**1.72**	59901	> 1800		> 1800	
29	no	**2.56**	48552	> 1800		> 1800	
30	no	**11.02**	129459	> 1800		> 1800	
31	no	**21.12**	187714	> 1800		> 1800	
32	no	**41.78**	296308	> 1800		> 1800	
33	no	**139.09**	198997	> 1800		456.16	2442
34	yes	37.69	95608	> 1800		**7.01**	22
35	no	**363.86**	354655	> 1800		> 1800	
36	yes	**464.60**	531818	> 1800		**7.40**	24
37	no	1051.50	1703756	1524.01	6031	**168.72**	2250
38	no	> 1800		117.09	878	32.36	314

Table 6.3: Evaluating the lookahead rule without restriction, and restricted to justified gates.

#	**VSIDS (no lookahead)**		**LH dec. heur. (4VSIDSLH)**		**LH dec. heur. (32VSIDSLH)**		**VSIDS (4VSIDSLH)**	
	time(s)	**# dec.**	**time(s)**	**# dec.**	**time(s)**	**# dec.**	**time(s)**	**# dec.**
1	**139.99**	280253	621.79	112713	1165.04	49668	216.90	84566
2	**104.00**	397522	498.96	118357	> 1800		346.24	179467
3	**483.01**	522928	> 1800		> 1800		1320.33	207529
4	132.80	189790	478.67	145720	1342.77	111748	**67.07**	37813
5	**250.61**	318747	753.58	148603	633.57	52924	714.93	165743
6	**83.23**	137902	> 1800		> 1800		> 1800	
7	381.01	484422	**155.71**	65248	395.16	42614	321.56	119266
8	**403.24**	425575	683.92	195653	574.21	68332	1239.33	322726
9	> 1800		**1771.75**	367783	> 1800		> 1800	
10	46.08	132492	71.12	47009	23.04	12476	**14.83**	24084
11	460.68	721780	200.15	139239	113.53	41715	**14.75**	19407
12	336.37	722126	274.34	188754	**115.02**	27689	1075.83	458660
13	165.92	323037	72.95	63261	101.52	33169	**80.82**	80738
14	246.14	632974	**171.31**	123191	506.02	74109	492.31	223768
15	**809.52**	1215931	1137.67	401268	1141.74	152110	810.77	368161
16	**121.60**	224553	374.15	100657	1292.91	75315	168.13	64655
17	**57.86**	130666	258.98	80200	406.73	41080	86.31	47720
18	**1024.70**	639245	1151.19	139266	1309.08	57538	1197.46	195685
19	**211.44**	506279	1795.42	376864	> 1800		864.54	413699
20	410.38	707957	508.12	170258	**291.25**	127520	463.99	320680
21	**69.14**	242506	833.78	228820	991.72	121655	156.15	148617
22	44.11	222240	474.18	160989	699.13	124846	**38.84**	124514
23	**47.40**	238891	1360.20	221885	> 1800		67.53	135381
24	> 1800		> 1800		> 1800		> 1800	
25	**820.30**	4827722	1013.14	1389252	> 1800		1764.40	2225911
26	**14.30**	164324	19.12	48710	43.14	35372	21.48	62045
27	**122.89**	993259	134.03	252281	382.86	213533	217.64	458298
28	**1.72**	59901	25.57	97238	86.16	111783	6.47	52488
29	**2.56**	48552	16.79	31455	44.01	25921	5.18	19841
30	**11.02**	129459	124.22	100586	592.63	169881	25.57	68366
31	**21.12**	187714	160.42	128993	1020.15	229483	31.41	80824
32	**41.78**	296308	435.30	176575	> 1800		94.95	180931
33	139.09	198997	**76.53**	22086	139.72	8085	159.37	54250
34	**37.69**	95608	163.29	47591	1517.17	43069	55.14	36981
35	363.86	354655	**139.46**	35990	332.53	14963	401.78	82560
36	464.60	531818	79.38	31174	> 1800		**51.27**	32471
37	1051.50	1703756	**1003.61**	218510	> 1800		> 1800	
38	> 1800		**1584.44**	304148	> 1800		> 1800	
	sums over solved instances							
	9121.66	19009889	18623.24	6480327	15260.81	2066598	12593.28	7097845
	runtime lower bound all instances							
	> 14521.66		>24023.24		> 36860.81		> 21593.81	
	instances solved							
	35		35		26		33	

Table 6.4: Evaluating lookahead rule combined with VSIDS scheme.

4VSIDSLH together with the normal VSIDS decision heuristic instead of the lookahead decision heuristic.

Again, all variations of the lookahead rule lead to worse overall performance, while the number of decisions is reduced at the same time. It is interesting to note that in another experimental run (not shown in the table), where we combined a 2VSIDSLH scheme with the VSIDS decision heuristic, the overall number of decisions was nearly halved compared to the VSIDS baseline. This might indicate that the second-highest scoring VSIDS literal quite often leads to conflicts.

While the implementation of the lookahead rule in BattleAx^3 is rather inefficient for small lookaheads, it still seems unlikely that the lookahead heuristic can be used successfully to make a solver faster or more robust, even if highly efficient implementation techniques are employed. Lookahead seems to be a heuristic that is mainly useful if the nature of the search is static, i.e., if a solver is not able to leave fruitless regions of the search space dynamically. It is likely that the benefits of the lookahead rule are negated by non-chronological backtracking and restarts.

6.3 Restarting Schemes

Restarts are among the most important extensions to the basic DLL framework and speed up the solving process significantly on average. Usually, restarts are triggered after a certain amount of conflicts or decisions. One of the considerations when implementing restarts is deciding on the sequence of such restart limits. In earlier implementations restart limit sequences are used which are monotonically rising. More recent implementations use very aggressive restarting schemes, which alternate long and very short runs. In addition, phase saving has been introduced as a means of recovering some of the progress that is lost during restarts.

In Table 6.5, a comparison of different restarting schemes is given, together with an evaluation of phase saving. We compare a geometric restarting scheme similar to the MiniSAT scheme [39], the nested scheme used in PicoSAT [5], and the scheme used in RSat 2.0 [36] that is based on the Luby sequence [31]. In each case, we have evaluated the impact of phase saving.

Surprisingly, the newer restarting schemes yield worse results than a simple geometric scheme on our benchmark set. Furthermore, we note a slight decrease in efficiency when combining phase saving with the geometric scheme. On both of the new non-monotonic restarting schemes, phase saving yields an efficiency increase. The results seem to indicate that phase saving does not in general increase the efficiency of restarts as was suggested by Biere [5], but instead reduces some of the negative side effects of very aggressive restarting schemes.

6.4 Comparing BattleAx^3 with other SAT Solvers

We have compared our implementation BattleAx^3 with both BCSat [26], the original implementation of the BC tableau procedure, and a modern CNF-based solver, MiniSAT

#	sat	**geom.**	**geom. ph.sav.**	**nested**	**nested ph.sav.**	**Luby**	**Luby ph.sav.**
1	no	**139.99**	236.26	190.76	224.93	181.00	181.51
2	yes	104.00	100.53	85.76	**63.52**	69.30	107.28
3	no	483.01	567.63	438.60	**349.04**	528.79	401.91
4	yes	132.80	420.45	**28.97**	52.16	244.51	538.25
5	yes	250.61	1189.97	1571.93	231.07	37.73	**19.97**
6	yes	**83.23**	965.30	> 1800	> 1800	1752.13	122.04
7	no	381.01	**237.65**	660.81	455.05	337.86	325.33
8	no	**403.24**	721.46	1580.75	1148.58	1122.67	828.72
9	no	> 1800	**1772.77**	> 1800	> 1800	> 1800	> 1800
10	yes	46.08	**5.67**	57.55	20.41	15.81	40.59
11	yes	460.68	67.42	114.28	140.36	81.40	**31.14**
12	yes	336.37	915.26	1553.89	371.12	1237.94	**202.50**
13	no	165.92	**97.69**	316.29	177.13	200.94	183.97
14	no	**246.14**	383.93	672.08	444.04	341.58	401.79
15	no	809.52	**732.03**	> 1800	1552.16	1503.90	1321.14
16	no	121.60	219.63	144.31	182.07	**85.60**	162.21
17	yes	57.86	159.02	25.65	**20.42**	91.66	44.65
18	no	1024.70	> 1800	379.15	**298.94**	471.60	509.32
19	no	211.44	313.58	147.15	158.90	192.59	**97.86**
20	yes	410.38	604.93	177.03	159.31	**58.33**	148.37
21	no	69.14	86.74	54.86	69.44	48.96	**43.08**
22	yes	44.11	42.89	73.81	83.92	79.79	**21.03**
23	no	47.40	73.61	52.92	46.43	**40.36**	51.20
24	no	> 1800	> 1800	> 1800	> 1800	> 1800	> 1800
25	no	820.30	**800.58**	> 1800	> 1800	> 1800	> 1800
26	no	**14.30**	18.26	140.57	141.65	56.95	48.99
27	no	122.89	**122.26**	1519.39	1788.02	302.05	481.56
28	no	**1.72**	1.97	2.53	2.96	3.38	4.28
29	no	**2.56**	2.97	3.42	3.60	3.54	4.10
30	no	**11.02**	14.56	20.28	18.16	15.05	15.35
31	no	21.12	28.04	24.21	22.42	17.97	**16.01**
32	no	**41.78**	42.99	84.30	74.98	47.08	48.08
33	no	139.09	110.19	116.88	108.63	128.57	**66.15**
34	yes	37.69	**4.98**	48.23	243.48	77.90	80.56
35	no	363.86	280.94	240.56	207.85	249.65	**146.34**
36	yes	464.60	162.06	**17.26**	207.38	28.03	291.41
37	no	**1051.50**	1232.13	1499.22	1139.09	> 1800	1532.31
38	no	> 1800	> 1800	> 1800	> 1800	> 1800	> 1800
sums over solved instances							
		9121.66	12736.35	12043.4	10207.22	9654.62	8519.00
instances solved							
		35	35	32	33	33	34
lower bound all instances							
		14521.66	18136.35	22843.4	19207.22	18654.62	15719.00

Table 6.5: Evaluating runtimes of restarting schemes and phase saving.

1.14 [39]. For the circuit-to-CNF translation we have used the tool bc2cnf by Tommi Juntilla, with the option "–nosimplify". For BattleAx^3, we have used the VSIDS decision heuristic and a geometric restarting scheme without phase saving.

A comparison of runtimes is presented in Table 6.6. It can easily be seen that BattleAx^3 significantly outperforms the original BC tableau procedure. While we are not able to surpass the performance of MiniSAT, the results indicate that our framework is competitive with the standard DLL procedure. It must be noted that BattleAx^3 is a prototype that lacks the extensive fine-tuning of CNF-based solvers. We believe that with efficient implementation techniques and through the introduction of structure-based extensions to the search, a circuit-based solver could eventually outperform a standard DLL solver such as MiniSAT.

Instance name	BattleAx^3	BCSat	MiniSAT
1394-4-3.p1neg.k10.unsat	139.99	>1800	**78.62**
1394-4-3.p1neg.k11.sat	104.00	170.58	**41.93**
1394-5-2.p0neg.k13.unsat	483.01	>1800	**72.60**
atree.sat.34.0	132.80	580.47	**49.20**
atree.sat.36.50	**250.61**	1135.88	274.05
atree.sat.38.100	**83.23**	>1800	225.83
atree.unsat.32.0	381.01	237.29	**72.38**
atree.unsat.34.50	403.24	769.67	188.13
atree.unsat.36.100	>1800	>1800	**450.62**
braun.sat.32.0	46.08	15.53	**9.53**
braun.sat.34.50	460.68	145.35	**48.43**
braun.sat.36.100	336.37	**36.96**	256.97
braun.unsat.32.0	165.92	38.23	**34.17**
braun.unsat.34.50	246.14	**61.67**	108.74
braun.unsat.36.100	809.52	135.17	**266.38**
brp.ptimonegnv.k23.unsat	121.60	>1800	**20.01**
brp.ptimonegnv.k24.sat	57.86	>1800	**6.77**
csmacd.p0.k16.unsat	1024.70	>1800	**106.71**
dme3.ptimo.k61.unsat	**211.44**	>1800	254.57
dme3.ptimo.k62.sat	410.38	>1800	**127.14**
dme3.ptimonegnv.k58.unsat	**69.14**	>1800	123.27
dme3.ptimonegnv.k59.sat	**44.11**	>1800	426.18
dme5.ptimo.k65.unsat	**47.40**	>1800	902.16
dp_12.i.k10.unsat	>1800	>1800	**90.46**
eq-test.atree.braun.10.unsat	820.30	301.66	**290.85**
eq-test.atree.braun.8.unsat	14.30	10.05	**8.76**
eq-test.atree.braun.9.unsat	122.89	59.48	**36.91**
fvp-unsat.2.0.3pipe.1	1.72	>1800	**0.37**
fvp-unsat.2.0.3pipe_2_ooo.1	2.56	>1800	**1.24**
fvp-unsat.2.0.4pipe_1_ooo.1	11.02	>1800	**2.22**
fvp-unsat.2.0.4pipe_2_ooo.1	21.12	>1800	**9.80**
fvp-unsat.2.0.5pipe_1_ooo.1	**41.78**	>1800	43.33
key_4.p.k28.unsat	139.09	56.95	**14.83**
key_4.p.k37.sat	**37.69**	>1800	159.82
key_5.p.k29.unsat	363.86	278.40	**44.68**
key_5.p.k37.sat	464.60	>1800	**39.24**
mmgt_4.i.k15.unsat	1051.50	>1800	**44.69**
q_1.i.k18.unsat	>1800	>1800	**36.87**
# instances solved	35	16	38
# runtime sum solved instances	9021.66	4033.34	4968.46
# full runtime sum	> 14521.66	> 25633.34	4968.46

Table 6.6: Comparing BattleAx^3 runtimes with BCSat and MiniSAT.

Chapter 7

Conclusion

In this thesis, we have presented an overview of the propositional satisfiability problem, describing techniques from CNF-SAT and approaches for solving the satisfiability problem in non-clausal instances and circuits. We have presented the BC tableau procedure for solving constrained circuit instances, and we have shown how a BC-based procedure can be implemented using a generalized DLL framework and extended with many of the techniques found in CNF-based SAT.

7.1 Evaluating Results

We have described BattleAx^3, a prototypical implementation of a BC-based procedure, and we have given an evaluation of a number of techniques on a set of benchmark circuits. Our extended solver performs significantly better than the original implementation of the tableau procedure described in Junttila and Niemelä [26], but is still slightly less efficient overall than a state-of-the-art CNF-based solver.

We were not able to achieve any speedups by introducing the lookahead rule used in the original implementation BCSat [26]. We have evaluated restricting the application of the lookahead rule according to a number of heuristics, and managed to increase its robustness, but the baseline implementation was still found to significantly outperform any application of lookahead. It seems that the increased mobility that is a result of non-chronological backtracking and restarts makes such an expensive heuristic unnecessary. Still, it is interesting to note that the lookahead heuristic leads to a significant decrease in decisions. On some instances, the lookahead heuristic manages to save two orders of magnitude on the numbers of decisions.

Comparing the two second-order decision heuristics VSIDS and BerkMin shows that the former slightly outperforms the latter on our benchmark set. We have encountered an interesting effect concerning average search depth and clause length that suggests that the two seemingly closely related decision heuristics actually differ considerably in the way in which they explore the search space. Analyzing this effect in more detail may yield important insights into the way the decision heuristic interacts with clause learning and the overall exploration of the search space.

7.2 Open Questions and Research Opportunities

In this thesis, a number of questions remain unexplored. The benchmark set which we have used to evaluate BattleAx^3 makes sparse use of the complex gate types that are supported by our solver, such as *card* gates and *even/odd* gates, but instead relies heavily on simple *and* and *or* gates, for which our counter-based deduction is admittedly suboptimal. It would be interesting to see how BattleAx^3 would perform on a benchmark set that made more extensive use of complex gate types.

On the other hand, speedups could be gained by extending our solver with a mix of different deduction techniques for different gate types. For some gates, such as *card* gates, our counter-based implementation is likely to be efficient, but other gate types could profit significantly from watched-input schemes akin to those presented in Wu et al. [51].

An area we have not explored at all is the choice between multiple implication graphs during the search. Contrary to CNF-based SAT (except for the lazy-assignment implementations discussed in Biere [3]), a number of reasons may be identified when applying a tableau rule. If, for example, we propagate a false input assignment in an *and* gate, we may choose as a reason any of the inputs that are assigned to false at this point. It is possible that the search process could be sped up by heuristically choosing between such reasons. With such a heuristic it may be possible, for example, to set up a conflict graph that allows for wide backjumps on average.

Another area where we suspect possible improvements is in the way learning is performed. In our implementation, we essentially perform CNF-based learning, which can be viewed as the addition of constrained *or* gates to the circuit. Such added gates are essentially flat, i.e., all of their outputs are primary outputs. It is possible that the learned information could be more concisely represented by extending the original circuit with more complex gate structures. In such a way, learned information may need to be purged less often than in the case CNF-based solvers in order for the solver to stay efficient.

While evaluating the VSIDS and BerkMin strategies, we have encountered an interesting effect concerning average search depth and the average length of conflict clauses. It may be interesting to analyze this effect more closely.

As a last point, it is interesting to note that it is easy to extend the BC-based implementation with new gate types. One the one hand, such gate types could encode any complex Boolean function where deduction might be handled more efficiently directly than by using an encoding with logical primitives (e.g., gates that perform binary arithmetic operations). On the other hand, it might prove interesting to use such general gate types as interfaces to external solvers for domains that are not easily expressible in propositional logic. As an example consider an instance of SAT-based planning. Here, a gate could represent an interface to an external domain-specific planner. In this way, problems could be solved that require a combination of abstract logical planning and domain specific planning.

7.3 Concluding Remarks

In this thesis, we have presented a framework for an efficient tableau-based solver. Our prototype does not surpass the efficiency of state-of-the-art CNF-based solvers, which is not surprising when considering the amount of effort that has gone into the area of CNF-based SAT. Nevertheless, our benchmarks indicate that the general framework of the extended tableau procedure is competitive with the standard DLL procedure.

Given proper implementation techniques and efficient structure-based heuristics it is likely that circuit-based SAT solving could be significantly sped up. With attention shifting increasingly towards solving non-clausal SAT instances, we believe that circuit-based solvers may soon outperform CNF-based solvers on practical instances.

Bibliography

[1] M. D. Agostino and M. Mondadori. The taming of the cut. Classical refutations with analytic cut. *Journal of Logic and Computation*, 4(3):285–319, 1994.

[2] R. J. Jr. Bayardo and R. C. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of the 14th National Conference on Artificial Intelligence (AAAI)*, pages 203–208. AAAI Press / The MIT Press, 1997.

[3] A. Biere. The evolution from LIMMAT to NANOSAT. Technical Report 444, Dept. of Computer Science, ETH Zurich, 2004.

[4] A. Biere. Formal systems 2 - lecture notes. `http://fmv.jku.at/fs2/`, 2006.

[5] A. Biere. PicoSAT essentials. *Journal on Satisfiability, Boolean Modeling and Computation*, 4:75–97, 2008.

[6] A. Biere. Resolve and expand. In *Proceedings of the 7th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 238–246. Online Proceedings, 2004.

[7] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *Proceedings of the 36th ACM/IEEE Conference on Design Automation (DAC)*, pages 317–320. ACM, 1999.

[8] T. Boy de la Tour. An optimality result for clause form translation. *Journal of Symbolic Computation*, 14(4):283–301, 1992.

[9] P. Chatalic and L. Simon. ZRES: The old Davis-Putnam procedure meets ZBDD. In *Proceedings of the 17th International Conference on Automated Deduction (CADE)*, volume 1831 of *Lecture Notes in Computer Science*, pages 449–454. Springer, 2000.

[10] S. A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the 3rd annual ACM Symposium on Theory of Computing (STOC)*, pages 151–158. ACM, 1971.

[11] J. M. Crawford and L. D. Auton. Experimental results on the crossover point in satisfiability problems. In *Proceedings of the 11th National Conference on Artificial Intelligence (AAAI)*, pages 21–27. AAAI Press, 1993.

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

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

[14] R. Drechsler, T. A. Juntilla, and I. Niemelä. *Handbook of Satisfiability*, chapter Non-Clausal SAT and ATPG (draft June 17, 2008). IOS Press - to be published, 2008.

[15] N. Eén and A. Biere. Effective preprocessing in SAT through variable and clause elimination. In *Proceedings of the 8th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, volume 3569 of *Lecture Notes in Computer Science*, pages 61–75. Springer, 2005.

[16] N. Eén and N. Sörensson. MiniSAT v2. 0 (beta). *Solver description, SAT Race*, 2006.

[17] U. Egly. *Automated Deduction. A basis for applications. Vol. 1*, chapter Cuts in Tableaux. Kluwer Academic Publishers, 1998.

[18] M. K. Ganai, P. Ashar, A. Gupta, L. Zhang, and S. Malik. Combining strengths of circuit-based and CNF-based algorithms for a high-performance SAT solver. In *Proceedings of the 39th Conference on Design automation (DAC)*, pages 747–750. ACM, 2002.

[19] E. Goldberg and Y. Novikov. BerkMin: A fast and robust Sat-solver. In *Proceedings of the 2002 Conference on Design, Automation and Test in Europe (DATE)*, pages 142–149. IEEE, 2002.

[20] C. P. Gomes, B. Selman, and H. Kautz. Boosting combinatorial search through randomization. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI)*, pages 431–437. AAAI Press / The MIT Press, 1998.

[21] A. Gupta, A. Gupta, Z. Yang, and P. Ashar. Dynamic detection and removal of inactive clauses in SAT with application in image computation. In *Proceedings of the 38th Conference on Design Automation (DAC)*, pages 536–541. ACM, 2001.

[22] J. Huang. The effect of restarts on the efficiency of clause learning. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 2318–2323. Online Proceedings, 2007.

[23] M. Järvisalo and I. Niemelä. The effect of structural branching on the efficiency of clause learning SAT solving: An experimental study. *Journal of Algorithms*, (1–3): 90–113, 2008.

[24] T. A. Juntilla. A file format for constrained Boolean circuits (accessed November 11, 2008). `http://www.tcs.hut.fi/~tjunttil/bcsat/`.

[25] T. A. Juntilla. personal communication, June 25 2008.

[26] T. A. Junttila and I. Niemelä. Towards an efficient tableau method for Boolean circuit satisfiability checking. In *Proceedings of the 1st International Conference on Computational Logic (CL)*, volume 1861 of *Lecture Notes in Computer Science*, pages 553–567. Springer, 2000.

[27] D. Kröning and O. Strichman. *Decision Procedures*, chapter Decision Procedures for Propositional Logic, pages 25–57. Springer, 2008.

[28] A. Kuehlmann and F. Krohm. Equivalence checking using cuts and heaps. In *Proceedings of the 34th annual Conference on Design Automation (DAC)*, pages 263–268. ACM, 1997.

[29] A. Kuehlmann, M. K. Ganai, and V. Paruthi. Circuit-based Boolean reasoning. In *Proceedings of the 38th Conference on Design Automation (DAC)*, pages 232–237. ACM, 2001.

[30] F. Lu, L. C. Wang, K. T. T. Cheng, J. Moondanos, and Z. Hanna. A signal correlation guided Circuit-SAT solver. *Journal of Universal Computer Science*, 10 (12):1629–1654, 2004.

[31] M. Luby, A. Sinclair, and D. Zuckerman. Optimal speedup of Las Vegas algorithms. In *Proceedings of the 2nd Israel Symposium on the Theory of Computing Systems (ISTCS)*, pages 128–133. IEEE, 1993.

[32] J. P. Marques-Silva. The impact of branching heuristics in propositional satisfiability algorithms. In *Proceedings of the 9th Portuguese Conference on Artificial Intelligence (EPIA)*, volume 1695 of *Lecture Notes in Computer Science*, pages 62–74. Springer, 1999.

[33] J. P. Marques-Silva and K. A. Sakallah. GRASP: A new search algorithm for satisfiability. In *Proceedings of the 1996 IEEE/ACM international Conference on Computer-aided Design (ICCAD)*, pages 220–227. IEEE, 1996.

[34] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference (DAC)*. ACM, 2001.

[35] K. Pipatsrisawat and A. Darwiche. A lightweight component caching scheme for satisfiability solvers. In *Proceedings of the 10th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, volume 4501 of *Lecture Notes in Computer Science*, pages 294–299. Springer, 2007.

[36] K. Pipatsrisawat and A. Darwiche. RSat 2.0: SAT solver description. *Solver description, SAT competition*, 2007.

[37] D. A. Plaisted and S. Greenbaum. A structure-preserving clause form translation. *Journal of Symbolic Computation*, 2(3):293–304, 1986.

[38] L. G. Silva, L. M. Silveira, and J. P. Marques-Silva. Algorithms for solving Boolean satisfiability in combinational circuits. In *Proceedings of the 1999 Conference on Design, Automation and Test in Europe (DATE)*, pages 526–530. IEEE, 1999.

[39] N. Sörensson and N. Eén. MiniSat v1.13 - a SAT solver with conflict-clause minimization. *Solver Description, SAT 2005*, 2005.

[40] R. M. Stallman and G. J. Sussman. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence*, 9(2):135–196, 1977.

[41] P. Stephan, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Combinational test generation using satisfiability. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 15(9):1167–1176, 1996.

[42] S. Subbarayan and D. K. Pradhan. NiVER: Non-increasing variable elimination resolution for preprocessing SAT instances. In *Proceedings of the 7th International Conference on Theory And Applications of Satisfiability Testing (SAT)*. Online Proceedings, 2004.

[43] The SAT 2005 Competition. `http://www.satcompetition.org/2005/`, 2005.

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

[45] The SAT Race 2008.
`http://www-sr.informatik.uni-tuebingen.de/sat-race-2008/`, 2008.

[46] C. Thiffault, F. Bacchus, and T. Walsh. Solving non-clausal formulas with DPLL search. In *Prooceedings of the 7th International Conference on Theory and Applications of Satisfiability Testing*. Online Proceedings, 2004.

[47] G. S. Tseitin. On the complexity of derivation in propositional calculus. *Studies in Constructive Mathematics and Mathematical Logic*, 2:115–125, 1968.

[48] M. N. Velev. Exploiting signal unobservability for efficient translation to CNF in formal verification of microprocessors. In *Proceedings of the 2004 Conference on Design, Automation and Test in Europe (DATE)*, pages 266–271. IEEE, 2004.

[49] M. N. Velev. Encoding global unobservability for efficient translation to SAT. In *Proceedings of the 7th International Conference on Theory and Applications of Satisfiability Testing (SAT)*. Online Proceedings, 2004.

[50] M. N. Velev. Comparison of schemes for encoding unobservability in translation to SAT. In *Proceedings of the 2005 Conference on Asia South Pacific Design Automation (ASP-DAC)*, pages 1056–1059. ACM, 2005.

[51] C. A. Wu, T. H. Lin, C. C. Lee, and C. Y. R. Huang. QuteSAT: A robust circuit-based SAT solver for complex circuit structure. In *Proceedings of the 2007 Conference on Design, Automation and Test in Europe (DATE)*, pages 1313–1318. ACM, 2007.

[52] H. Zhang and M. E. Stickel. Implementing the Davis-Putnam method. *Journal of Automated Reasoning*, 24(1/2):277–296, 2000.

[53] L. Zhang and S. Malik. Cache performance of SAT solvers: a case study for efficient implementation of algorithms. In *Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, volume 2919 of *Lecture Notes in Computer Science*, pages 287–298. Springer, 2003.

[54] L. Zhang and S. Malik. The quest for efficient Boolean satisfiability solvers. In *Proceedings of the 18th International Conference on Automated Deduction (CADE)*, volume 2392 of *Lecture Notes in Computer Science*, pages 295–313. Springer, 2002.

[55] L. Zhang, C. F. Madigan, M. H. Moskewicz, and S. Malik. Efficient conflict driven learning in a Boolean satisfiability solver. In *Proceedings of the 2001 International Conference on Computer-Aided Design (ICCAD)*, pages 279–285. IEEE, 2001.