Towards an Efficient Tableau Method for Boolean Circuit Satisfiability Checking

Tommi A. Junttila and Ilkka Niemelä

Helsinki University of Technology
Dept. of Computer Science and Engineering
Laboratory for Theoretical Computer Science
P.O.Box 5400, FIN-02015 HUT, Finland
{Tommi.Junttila,Ilkka.Niemela}@hut.fi

Abstract. Boolean circuits offer a natural, structured, and compact representation of Boolean functions for many application domains. In this paper a tableau method for solving satisfiability problems for Boolean circuits is devised. The method employs a direct cut rule combined with deterministic deduction rules. Simplification rules for circuits and a search heuristic attempting to minimize the search space are developed. Experiments in symbolic model checking domain indicate that the method is competitive against state-of-the-art satisfiability checking techniques and a promising basis for further work.

1 Introduction

Propositional satisfiability checkers have been applied successfully to many interesting domains such as planning [11] and model checking of finite state systems [1, 3, 2]. The success has built on recent significant advances in the performance of SAT checkers based both on stochastic local search algorithms and on complete systematic search.

In this paper we are interested in developing SAT checking methodology especially for symbolic model checking purposes. Most work on symbolic model checking [6] has been based on binary decision diagrams (BDDs) [5]. However, BDD-based methods suffer from the fact that a BDD representation of a Boolean expression can require exponential space. Recent research has shown that this problem can be overcome by using state-of-the-art SAT checking methods which work in polynomial space in the size of the input [1, 3, 2].

Most successful satisfiability checkers assume that the input formulae are in conjunctive normal form (CNF). This sometimes makes efficient modeling of an application challenging because natural non-clausal formalizations can lead to significant blow-up when the formulae are transformed to CNF. An example of the CNF transformation problem is a formula of the form

$$(P_1 \wedge Q_1) \vee \cdots \vee (P_n \wedge Q_n)$$

whose equivalent CNF is of exponential size. If it is enough to preserve satisfiability, the size of the CNF can be decreased to linear by introducing a new atom

J. Lloyd et al. (Eds.): CL 2000, LNAI 1861, pp. 553–567, 2000.

for each conjunction and transforming the formula to

$$(R_1 \vee \cdots \vee R_n) \wedge (R_1 \leftrightarrow (P_1 \wedge Q_1)) \wedge \cdots \wedge (R_n \leftrightarrow (P_n \wedge Q_n))$$

whose CNF is

$$(R_1 \vee \cdots \vee R_n) \wedge \cdots \wedge (R_n \vee \neg P_n \vee \neg Q_n) \wedge (\neg R_n \vee P_n) \wedge (\neg R_n \vee Q_n) \quad .$$

Notice, however, that now the number of atomic formulae has increased by n and the search space (and running time) of typical SAT checkers could increase exponentially.

In this paper we study an alternative approach to solving propositional satisfiability problems which is not based on representing the input in CNF but as a *Boolean circuit*. This allows a compact and natural representation in many domains. Using Boolean circuits as the input format makes it possible to simplify the representation by sharing common subexpressions and by preserving natural structures and concepts of the domain.

Our idea is to combine advantages of a compact representation based on Boolean circuits and polynomial space requirements of CNF-based search procedures and devise a satisfiability checking algorithm for Boolean circuits, i.e., a procedure for finding truth assignments for a circuit given some constraints on its output values or for determining that none such exists.

There is a lot of previous research on theorem proving and satisfiability checking methods working with arbitrary (non-clausal) formulae. The work is mainly based on tableaux and sequent calculi, see e.g. [13] for a technique to make this approach more amenable to real applications using simplification methods. A commercial SAT-checking system, Prover [4], is also working with non-CNF input. Also SAT checking systems basically working with CNF input have been extended to handle more general formulae. This has been done both for complete SAT checkers, e.g. in [8], as well as for local search methods [10, 17].

In this paper we develop a tableau method that works directly with Boolean circuits. Instead of standard (cut free) tableau techniques we employ a direct cut rule combined with deterministic (non-branching) deduction rules. The aim is to achieve high performance and to avoid some computational problems in cut free tableaux [7]. In order make the method more efficient we devise simplification rules and a search heuristic which attempts to minimize the search space of the algorithm. The heuristic is inspired by the search technique used in a system for computing stable models [15, 18]. Experimental results indicate that the cut-based tableau method combined with suitable deduction and simplification rules and the search space minimizing heuristic has promising performance, e.g., in symbolic model checking applications.

The rest of the paper is structured as follows. We start by introducing Boolean circuits. Then we develop a tableau method for circuits. In Section 4 we identify transformation rules for simplifying circuits and then we describe an experimental implementation of the tableau method. Section 6 provides a simple translation of circuits to CNF used in the experiments presented in the following section where our tableau method is compared to state-of-the-art satisfiability checkers using symbolic model checking benchmarks.

2 Boolean Circuits

A *Boolean circuit* C is an acyclic directed graph where the nodes are called the *gates* of C. The gates with no outgoing edges are the *output gates* and the gates with no incoming edges and no Boolean function are the *input gates* of C. Each non-input gate is associated with a Boolean function and "calculates" its value from the values of its children. In this paper we represent Boolean circuits as *Boolean equation systems*. Such systems offer a convenient way of writing down circuits and of describing transformations on them. For more on Boolean circuits, see e.g. [16].

Given a finite set $\mathcal{V}$ of Boolean variables, a *Boolean equation system* (a *system* for short) $\mathcal{S}$ over $\mathcal{V}$ is a set of equations of the form $v = f(v_1, \ldots, v_k)$, where $v, v_1, \ldots, v_k \in \mathcal{V}$ and f is an arbitrary Boolean function. Boolean circuits can be seen as Boolean equation systems of a certain form where each variable has at most one equation and the equations are not recursive. More precisely this can be characterized as follows. Given a Boolean equation system $\mathcal{S}$ over $\mathcal{V}$ such that for each variable $v \in \mathcal{V}$ there is at most one equation in $\mathcal{S}$, we define the directed graph $G(\mathcal{S}) = \langle \mathcal{V}, E \rangle$, where $E = \{\langle v', v \rangle \mid v = f(\ldots, v', \ldots) \in \mathcal{S}\} \subseteq \mathcal{V} \times \mathcal{V}$. The graph $G(\mathcal{S})$ describes the variable dependencies in $\mathcal{S}$ and if $G(\mathcal{S})$ is acyclic, then $G(\mathcal{S})$ can be seen as a Boolean circuit. See Fig. 1 for an example. The variables of $\mathcal{S}$ correspond to the gates of the circuit and the variables for which there is no equation are the input gates of the circuit. A variable defined by an equation of form $v = \top$ ($v = \bot$) in turn corresponds to a constant gate "true" ("false").

$$
\begin{aligned}
v_1 &= \mathsf{and}(v_2, v_3) \\
v_2 &= \mathsf{not}(v_4) \\
v_3 &= \mathsf{or}(v_4, v_5, v_6) \\
v_6 &= \bot
\end{aligned}
$$

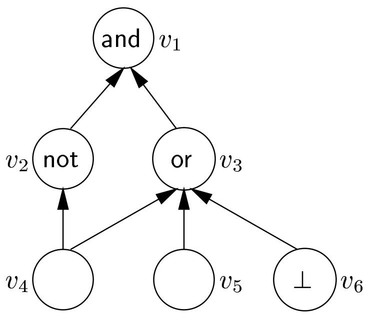

Fig. 1. A system over $\{v_1, \ldots, v_6\}$ and the corresponding Boolean circuit.

A *truth valuation* for $\mathcal{S}$ is a function $\tau : \mathcal{V} \to \{\text{true}, \text{false}\}$. Valuation τ is *consistent* if $\tau(v) = f(\tau(v_1), \ldots, \tau(v_k))$ holds for each equation $v = f(v_1, \ldots, v_k)$ in $\mathcal{S}$. A system $\mathcal{S}$ is *satisfiable* if there exists a consistent valuation for it. The question of whether a system is satisfiable is obviously an **NP**-complete problem under the plausible assumption that each Boolean function appearing in the system can be computed in deterministic polynomial time. However, note that each Boolean equation system describing a Boolean circuit has exactly 2^n

consistent truth assignments, where n is the number of input gates in the circuit (the system only describes the *structure* of the circuit). Therefore, in case of Boolean circuits, we are interested in the *constrained satisfaction problem*: given that variables in $c^+ \subseteq \mathcal{V}$ must be true and those in $c^- \subseteq \mathcal{V}$ false (the *constraints*), is there a consistent valuation that respects these constraints? Again, this is obviously an **NP**-complete problem.

In the rest of the paper we consider the class of Boolean circuits where the following Boolean functions are allowed in the gates (equations).

- $\top$ (a constant function) is always true. The constant $\bot$ is always false.
- $\mathsf{equiv}(v_1, \ldots, v_n)$ = true iff all v_i, $1 \leq i \leq n$, are true or all v_i, $1 \leq i \leq n$, are false.
- $\mathsf{or}(v_1, \ldots, v_n)$ = true iff at least one v_i, $1 \leq i \leq n$, is true.
- $\mathsf{and}(v_1, \ldots, v_n)$ = true iff all v_i, $1 \leq i \leq n$, are true.
- $\mathsf{even}(v_1, \ldots, v_n)$ = true iff an even number of v_is, $1 \leq i \leq n$, are true.
- $\mathsf{odd}(v_1, \ldots, v_n)$ = true iff an odd number of v_is, $1 \leq i \leq n$, are true.
- $\mathsf{not}(v)$ = true iff v is not true.

3 A Tableau Method

In this section we develop a tableau method for solving satisfiability problems for constrained Boolean circuits. A straightforward approach would be to interpret each equation $v = f(v_1, \ldots, v_k)$ as an equivalence $v \leftrightarrow f(v_1, \ldots, v_k)$. We could thus use a traditional tableau method [14] by setting (i) for each equation $v = f(v_1, \ldots, v_k)$ in $\mathcal{S}$ an entry $\mathbf{T}(v \leftrightarrow f(v_1, \ldots, v_k))$ and (ii) for each constraint $v \in c^+$ ($v \in c^-$) an entry $\mathbf{T}v$ ($\mathbf{F}v$) in the tableau root and then applying the standard tableau rules.

However, standard (cut free) tableau methods suffer from some computational problems [7]. In order to overcome these we use a tableau system that has an explicit cut rule while all the rest of the rules are deterministic. The basic rules are shown in Fig. 2. Note that the versions of rules obtained by commutativity of the operations are not shown, e.g., the following is a rule:

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

Given a system $\mathcal{S}$, the root of the tableau consists of the equations in $\mathcal{S}$ and the constraints. The rules appearing in Fig. 2 are then applied as in the standard tableau method. A branch in the tableau is *contradictory* if it contains both $\mathbf{F}v$ and $\mathbf{T}v$ entries for a variable in $v \in \mathcal{V}$. A branch is *complete* if it contains an $\mathbf{F}v$ or $\mathbf{T}v$ entry for each $v \in \mathcal{V}$ and no application of rules in Fig. 2(b)–(f) leads to contradiction.

Theorem 1. *The above tableau system is sound and complete in the sense that a complete branch gives a satisfying truth assignment for $\mathcal{S}$ while the absence of a complete branch indicates that the system is unsatisfiable.*

$$\frac{v \in \mathcal{V}}{\mathbf{T}v \,\big|\, \mathbf{F}v}$$

(a) The explicit cut rule

$$\frac{v = \top}{\mathbf{T}v} \qquad \frac{v = \bot}{\mathbf{F}v}$$

(b) Constant rules

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

(c) Negation rules

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

(d) "Up" rules for or and and

$$\frac{\begin{array}{c} v = \mathsf{equiv}(v_1, \ldots, v_k) \\ \mathbf{T}v_1, \ldots, \mathbf{T}v_k \end{array}}{\mathbf{T}v} \quad \frac{\begin{array}{c} v = \mathsf{equiv}(v_1, \ldots, v_k) \\ \mathbf{F}v_1, \ldots, \mathbf{F}v_k \end{array}}{\mathbf{T}v} \quad \frac{\begin{array}{c} v = \mathsf{equiv}(v_1, \ldots, v_k) \\ \mathbf{T}v_i,\ i \in \{1, \ldots, k\} \\ \mathbf{F}v_j,\ i \in \{1, \ldots, k\} \end{array}}{\mathbf{F}v}$$

(e) "Up" rules for equiv

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

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

(f) "Up" rules for even and odd

Fig. 2. Basic rules.

Notice that for Boolean circuits it would be sufficient to apply the cut rule to the input gates only: other gates are functionally fully dependent on input (and constant) gates. Therefore the values of all gates can be evaluated by using the rules in Fig. 2(b)–(f) once the values of input gates are assigned.

The size of a tableau depends essentially on the branching of the tableau, i.e., on the number of times that the *cut rule* in Fig. 2(a) is applied. In order to avoid the use of the cut rule we devise a set of additional rules which complement the basic rules. These rules given in Fig. 3 can be used in the tableau construction without affecting its soundness or completeness. In the following, the rules in Fig. 2(b)–(f) and Fig. 3 are called the *deterministic deduction rules*.

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

(a) "Down" rules for not

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

(b) "Down" rules for or, and and equiv

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

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

(c) "Last undetermined child" rules for or, and and equiv

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

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

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

(d) "Last undetermined child" rules for even and odd

Fig. 3. Additional deduction rules.

Example 1. Consider the circuit in Fig. 1 and the constrained satisfaction problem where variable v_1 must be true. Below is a tableau solving this problem

$$
\begin{array}{ll}
1.\ v_1 = \mathsf{and}(v_2, v_3) & \\
2.\ v_2 = \mathsf{not}(v_4) & \\
3.\ v_3 = \mathsf{or}(v_4, v_5, v_6) & \\
4.\ v_6 = \bot & \\
5.\ \mathbf{T}v_1 & \\
6.\ \mathbf{F}v_6 & (4)
\end{array}
$$

$$\swarrow\ \searrow$$

$$
\begin{array}{llll}
7.\ \mathbf{T}v_4 & (\text{cut}) & 8.\ \mathbf{F}v_4 & (\text{cut}) \\
9.\ \mathbf{F}v_2 & (2,7) & 11.\ \mathbf{T}v_2 & (2,8) \\
10.\ \mathbf{F}v_1 & (1,9) & 12.\ \mathbf{T}v_3 & (1,5,11) \\
\quad\times & (5,10) & 13.\ \mathbf{T}v_5 & (3,6,8)
\end{array}
$$

where expressions 1–4 represent the circuit and expression 5 the constraint. For each other expression we give in parentheses the expressions from which it is derived using the tableau rules. Notice that the left hand branch (1–7,9,10) is contradictory and does not provide a solution but the right hand branch (1–6,8,11–13) is complete and yields a satisfying truth assignment where $\tau(v_1) = \tau(v_2) = \tau(v_3) = \tau(v_5) =$ true and $\tau(v_4) = \tau(v_6) =$ false.

The use of the cut rule can be further limited by employing stronger deterministic deduction rules. There is an interesting trade-off between the computational complexity involved in implementing a deduction rule and its ability to derive further truth values. We consider as an interesting compromise a deduction rule that we call *one-step lookahead.*

One-step lookahead: Consider a branch B and an expression $\mathbf{T}v$ ($\mathbf{F}v$). If a complementary pair of variables $\mathbf{T}w, \mathbf{F}w$ can be derived using the deterministic deduction rules from $B \cup \{\mathbf{T}v\}$ (from $B \cup \{\mathbf{F}v\}$), deduce $\mathbf{F}v$ ($\mathbf{T}v$).

For instance, in the example above one-step lookahead could be applied to the branch containing expressions 1–6 and for $\mathbf{T}v_4$. Now $\mathbf{T}v_1, \mathbf{F}v_1$ can be derived and, hence, $\mathbf{F}v_4$ can be deduced. After that the branch can be completed using the deterministic deduction rules and a solution is found without any cuts (branching). The one-step lookahead rule is similar to the failed literal rule [12] in Davis-Putnam procedures for CNF satisfiability checking and the lookahead rule in Smodels system [18] computing stable models. Notice that examining whether the lookahead rule is applicable for a given expression $\mathbf{T}v$ ($\mathbf{F}v$) can be done in linear time in the size of the branch (given appropriate implementation techniques). Hence, determining the applicability of the rule is more expensive than for the other deterministic deduction rules. However, the lookahead rule is quite powerful in decreasing the number of cut rule applications needed to determine the existence of a solution. Experimental results indicate that often the additional overhead is worth the effort.

4 Satisfiability Preserving Simplifications

In order to simplify the structure of a circuit, some efficiently implementable, simple satisfiability preserving simplifications can be applied to a circuit. Actually, some of these simplifications require that the value of a gate is assigned and should thus be applied to a constrained circuit (where $\mathbf{T}v$ and $\mathbf{F}v$ provide the information).

1. Common subexpressions can be *shared*, i.e., if a system has two similar equations, $v = f(v_1, \ldots, v_k)$ and $v' = f(v_1, \ldots, v_k)$, then $v' = f(v_1, \ldots, v_k)$ can be removed from the system and all the occurrences of v' are substituted with v.
2. If a gate has a child whose value is determined, the connection to the child can be removed by the rewriting rules shown in Fig. 4(b)–(c). Figure 4 also shows some other simplification rewriting rules. One simplification that deserves special attention is the "input gate under true equivalence"-simplification in Fig. 4(d). It can detect that an input gate is functionally fully depended on other gates and removes it.
3. A *cone of influence* reduction can be performed: if a variable is not constrained and no other equation refers to it (i.e. it is an output gate in the circuit), it can be removed, that is, gates that cannot influence constrained gates can be removed.

5 An Experimental Implementation

We have made an experimental implementation called BCSat [9] of the tableau method described in Sec. 3 for Boolean circuits. In the following we briefly discuss the implementation.

After parsing in the circuit, some simple preprocessing steps are applied to it. First, we set the constraints in the tableau root. We then apply the deterministic deduction rules, one-step lookahead and the satisfiability preserving simplifications of Sec. 4 until nothing new can be deduced. Naturally, if any step here leads to a contradiction, the circuit is unsatisfiable and the procedure is stopped.

After this the tableau is built one branch at a time using a chronological backtracking procedure. At each search level, we first apply the deterministic rules and the one-step lookahead rule as long as they produce new information. We then choose the next undetermined *cut variable* for which the cut rule is applied, after which the search branches to the next level. The cut variable is selected by using the following heuristic. For each undetermined variable v the following question is considered: if v is set to false (true), how many values of other undetermined variables can be deduced by using the deterministic rules? Call these numbers $v^{\perp}$ and $v^{\top}$, respectively. A variable for which $\min\{v^{\perp}, v^{\top}\}$ is largest is then selected as the cut variable. The reasoning behind this choice of cut variable is that choosing a maximum of the minimum minimizes the sum of the remaining search space (the number of still possible variable assigments) left

$$\frac{v = \mathsf{and}()}{v = \top} \qquad \frac{v = \mathsf{or}()}{v = \bot} \qquad \frac{v = \mathsf{equiv}()}{v = \top} \qquad \frac{v = \mathsf{even}()}{v = \top} \qquad \frac{v = \mathsf{odd}()}{v = \bot}$$

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

(a) Simplification rules for 0-ary and 1-ary gates

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

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

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

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

(b) "Determined child"-simplification rules for or, and, even and odd

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

(c) A "determined child"-simplification rule for equiv

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

$$\frac{\begin{array}{c} v = \mathsf{not}(v') \\ v' = \mathsf{not}(v'') \end{array}}{\begin{array}{c} v = v'' \\ v' = \mathsf{not}(v'') \end{array}} \qquad \frac{\begin{array}{c} v = \mathsf{and}(\ldots, v', \ldots, v_1, \ldots) \\ v_1 = \mathsf{not}(v') \end{array}}{\begin{array}{c} v = \bot \\ v_1 = \mathsf{not}(v') \end{array}} \qquad \frac{\begin{array}{c} v = \mathsf{or}(\ldots, v', \ldots, v_1, \ldots) \\ v_1 = \mathsf{not}(v') \end{array}}{\begin{array}{c} v = \top \\ v_1 = \mathsf{not}(v') \end{array}}$$

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

Fig. 4. Satisfiability preserving simplification *rewriting* rules for constrained circuits (an equation of form $v = v'$ means that occurrences of v are substituted with v').

in both search branches: if the number of undetermined variables is N, then after the cut and deterministic rules the search space left is $\mathcal{O}(2^{N-v^{\perp}}+2^{N-v^{\top}})$. This is minimized by our heuristic and we have thus chosen a *balancing* heuristic rather than a greedy one. As a small improvement we do not count the determined not-gates into $v^{\perp}$ or $v^{\top}$ since not is a fully deterministic operation w.r.t. to its only argument. Our experiments so far indicate that counting in all the variables when computing $v^{\perp}$ and $v^{\top}$ leads to smaller tableaux than when considering only the undetermined input variables.

The lookahead and computation of $v^{\perp}$ and $v^{\top}$ are implemented simply by first assigning an undetermined variable v to false and then applying the deterministic rules. The number $v^{\perp}$ is then stored and the effects of the assignment and application of deterministic rules are undone. The same procedure is repeated for true. If it is found out that assigning a variable to false (true) leads to a contradiction but assignment to true (false) does not, the variable is assigned to true (false), and the deterministic rules are applied. If both assignments lead to contradiction, backtracking to the previous search level occurs. This kind of lookahead and its use was inspired by the one used in the Smodels system [18].

6 Translating Circuits into CNF

In order to compare our tool to some satisfiability checkers requiring the input to be in CNF, we now present a very simple translation from Boolean circuits to CNF. We do not treat here equiv-, even- or odd-gates with more than 2 inputs (this would require more than a linear number of clauses or additional variables). Furthermore, the experimental cases we consider do not have such gates. The CNF translation is the conjunction of clauses obtained from the gates by the translation rules in Table 1. Input gates (variables with no definitions) are

Table 1. Boolean circuit to CNF translation rules.

Gate	CNF clause(s)
$v = \top$	v
$v = \perp$	$\neg v$
$v = \mathsf{not}(v_1)$	$(v \vee v_1) \wedge (\neg v \vee \neg v_1)$
$v = \mathsf{or}(v_1, \ldots, v_n)$	$(v \vee \neg v_1) \wedge \cdots \wedge (v \vee \neg v_n) \wedge (\neg v \vee v_1 \vee \cdots \vee v_2)$
$v = \mathsf{and}(v_1, \ldots, v_n)$	$(\neg v \vee v_1) \wedge \cdots \wedge (\neg v \vee v_n) \wedge (v \vee \neg v_1 \vee \cdots \vee \neg v_2)$
$v = \mathsf{even}(v_1, v_2)$ and $v = \mathsf{equiv}(v_1, v_2)$	$(v \vee v_1 \vee v_2) \wedge (v \vee \neg v_1 \vee \neg v_2) \wedge$ $(\neg v \vee v_1 \vee \neg v_2) \wedge (\neg v \vee \neg v_1 \vee v_2)$
$v = \mathsf{odd}(v_1, v_2)$	$(v \vee v_1 \vee \neg v_2) \wedge (v \vee \neg v_1 \vee v_2) \wedge$ $(\neg v \vee v_1 \vee v_2) \wedge (\neg v \vee \neg v_1 \vee \neg v_2)$

translated into $(v \vee \neg v)$. The constraints in the constrained satisfaction problem are simply translated into corresponding unit clauses (like the constant gates).

7 Some Experiments

We use the bounded model checking examples of Biere et al [1]. For each problem instance we use two different input sources. First one is the DIMACS CNF output produced directly by the bounded model checker tool **BMC** [1]. We will call this format **BMC**-CNF. We were also able to reconstruct Boolean circuits from the Prover output format files produced by **BMC**. These circuits were used as such or translated into CNF as described in Sec. 6.

We used the following solvers: BCSat described in this paper, CGrasp [8], Satz [12] and Sato [19]. BCSat and CGrasp both work on Boolean circuit input formats (we made a straightforward translation from our format to that of CGrasp). These tools were thus ran only on the Boolean circuit input. Since Satz and Sato expect DIMACS CNF as input format, we ran these tools in both **BMC**-CNF and CNF translated from circuits. All solvers were used "as is", no engineering work was put on trying to find suitable parameter settings. Unfortunately, we did not have access to the Prover tool [4].

The tests were run on 450 MHz Pentium machines running the Linux operating system. The times shown are the user times measured with the `time` command. The times do not include neither the generation of input files with the **BMC** tool nor translations between formats.

As the first test case we used the barrel shifter, with results shown in Table 2. The parameter $|r|$ in the first column indicates the number of registers in the shifter and also the number of time steps in **BMC**. The next two columns show the times of CNF solvers Sato and Satz when ran on **BMC**-CNF. The next four columns show the solver times when ran on unsimplified Boolean circuit input (translated into CNF in case of Sato and Satz). The last column shows the running time of the BCSat tool when allowed to make simplifications described in Sec. 4. The striking difference in the performance is due to the "input gate under true equivalence"-simplification in Fig. 4(d): BCSat finds out during the simplification that the circuit is not satisfiable and does thus not perform any actual search. This observation also explains the good results of the Prover tool as described in [1]: Prover simplifies the equivalences, too.

Table 2. Barrel shifter ($|r|$ = number of registers).

	BMC-CNF		Circuit, no red.				BCSat
$\|r\|$	Sato	Satz	BCSat	CGrasp	Sato	Satz	red.
4	0	0	0	0	5	0	0
5	13	465	302	135	$\geq$1h	44	0
6	73	$\geq$1h	$\geq$1h	$\geq$1h	-	224	0
7	280	-	-	-	-	1369	0
8	613	-	-	-	-	$\geq$1h	0
9	$\geq$1h	-	-	-	-	-	0
10	-	-	-	-	-	-	0

Table 3 shows our next example in which a counter-example of length k has to be found in a buggy design of a mutual exclusion algorithm under fairness. Unlike in other examples, the instances here are satisfiable. Again, the solvers are run on **BMC**-CNF, circuit input and then on simplified circuit input (the simplification times for solvers other than BCSat are not included in their running times).

Table 3. Counterexample for liveness in a buggy DME with 2 cells.

	BMC-CNF		Circuit, no red.				Circuit, red.			
k	Sato	Satz	BCSat	CGrasp	Sato	Satz	BCSat	CGrasp	Sato	Satz
10	0	1	104	17	1	1	4	4	0	0
11	1	3	3	28	3	≥1h	3	6	0	≥1h
12	0	6	4	61	5	4	3	9	0	≥1h
13	≥1h	≥1h	5	126	136	3	4	11	0	-
14	2	≥1h	6	152	6	3	6	23	0	-
15	249	-	223	150	152	4	6	21	0	-
16	≥1h	-	8	129	≥1h	5	13	36	7	-
17	≥1h	-	8	178	≥1h	5	8	94	1	-
18	-	-	13	201	-	6	10	57	3	-
19	-	-	10	1255	-	7	19	99	4	-
20	-	-	14	445	-	8	514	110	23	-
21	-	-	16	369	-	8	13	161	1	-
22	-	-	22	1253	-	10	17	190	≥1h	-
23	-	-	24	412	-	11	15	210	3185	-
24	-	-	26	891	-	11	19	349	23	-
25	-	-	27	867	-	11	20	666	2	-
26	-	-	30	2573	-	14	26	666	11	-
27	-	-	28	892	-	16	30	3091	≥1h	-
28	-	-	38	1356	-	24	34	2941	≥1h	-
29	-	-	34	937	-	35	34	2815	-	-
30	-	-	47	≥1h	-	46	38	3159	-	-

Our two last examples concern the same mutual exclusion algorithm, this time a correct one (thus there are no counter-examples and the instances are unsatisfiable). Table 4 shows results in the case of two users, parameterized w.r.t. the number of time steps. This means that we parameterize over the circuit depth since the greater the number of time steps, the greater the circuit depth. On the other hand, Table 5 depicts results when the number of time steps is kept as 10 but the number of users is parameterized. This corresponds to parameterization over circuit width. When comparing these two parameterization dimensions, we notice that the circuit depth seems to be a more crucial dimension for solver efficiency.

Admittedly, in order to draw any firm conclusions on the behavior of different solvers, more experiments should be conducted, especially on other types of Boolean circuits. However, we make some preliminary observations: (i) BCSat

Table 4. Liveness in DME with 2 cells, parameterized w.r.t. the number of time steps.

	BMC-CNF		Circuit, no red.				Circuit, red.			
k	Sato	Satz	BCSat	CGrasp	Sato	Satz	BCSat	CGrasp	Sato	Satz
10	1322	1	2	43	178	1	1	18	1	0
11	≥1h	1	3	159	≥1h	1	3	17	≥1h	0
12	≥1h	2	3	113	≥1h	2	4	48	12	1
13	-	5	5	193	-	3	5	68	7	1
14	-	15	6	280	-	4	6	72	27	3
15	-	52	9	380	-	8	8	95	6	3
16	-	≥1h	23	1859	-	10	12	105	159	7
17	-	≥1h	37	641	-	25	14	188	23	10
18	-	-	47	858	-	34	17	245	166	21
19	-	-	55	1010	-	78	343	428	≥1h	37
20	-	-	117	2010	-	32	524	400	≥1h	67
21	-	-	341	3457	-	211	1054	633	-	326
22	-	-	3461	1682	-	≥1h	≥1h	2496	-	≥1h
23	-	-	≥1h	3591	-	≥1h	≥1h	1274	-	≥1h
24	-	-	≥1h	3495	-	-	-	2254	-	-
25	-	-	-	2621	-	-	-	≥1h	-	-
26	-	-	-	≥1h	-	-	-	1962	-	-
27	-	-	-	≥1h	-	-	-	≥1h	-	-
28	-	-	-	-	-	-	-	2939	-	-
29	-	-	-	-	-	-	-	≥1h	-	-
30	-	-	-	-	-	-	-	≥1h	-	-

Table 5. Liveness in DME with 10 time steps, parameterized w.r.t. the number of cells.

	BMC-CNF		Circuit, no red.				Circuit, red.			
cells	Sato	Satz	BCSat	CGrasp	Sato	Satz	BCSat	CGrasp	Sato	Satz
2	1322	1	2	43	178	1	1	18	1	0
3	≥1h	4	4	82	≥1h	2	4	36	1	1
4	≥1h	18	7	340	≥1h	4	8	59	124	1
5	-	73	13	364	-	5	14	125	≥1h	1
6	-	≥1h	20	691	-	7	20	129	25	2
7	-	≥1h	28	1058	-	13	34	88	≥1h	6
8	-	-	35	1891	-	10	39	204	≥1h	3
9	-	-	47	1678	-	16	50	166	-	4
10	-	-	67	2053	-	20	63	178	-	5
11	-	-	81	2567	-	22	83	381	-	5
12	-	-	93	≥1h	-	31	98	416	-	7
13	-	-	101	≥1h	-	26	107	781	-	7
14	-	-	119	-	-	46	126	872	-	19
15	-	-	139	-	-	43	165	817	-	9
16	-	-	160	-	-	44	183	926	-	15
17	-	-	192	-	-	56	218	1059	-	12
18	-	-	232	-	-	94	239	1035	-	13
19	-	-	242	-	-	85	260	584	-	42
20	-	-	275	-	-	112	295	1653	-	19

and Satz seem to work quite similarly: this is probably because they both use lookahead and their heuristics are somewhat similar. (ii) All solvers seem to be a bit input syntax sensitive: there are cases when simplifications help but also counter-cases.

Finally, note that since our Boolean circuits were reconstructed from the output generated for the Prover tool, the circuits are probably not equal to those that would be generated should the **BMC** tool support Boolean circuit formalism directly. However, we assume that the circuits we have are quite close to those.

8 Conclusions

We have developed a tableau method for solving Boolean circuit satisfiability problems. The method works directly on Boolean circuits and does not require any clausal form translation of the circuit. Our method differs from standard tableau techniques. It uses an explicit cut rule together with deterministic (non-branching) deduction rules. In addition to typical deduction rules propagating truth values, our method employs a one-step lookahead rule which is computationally more expensive than standard propagation rules but which enables stronger propagation and reduces the need to use the cut rule. Furthermore, we identify simplification rules which preserve satisfiability but reduce the size and the form of a circuit. We have developed a prototype implementation of the method which applies the simplification rules and builds a tableau one branch at the time using backtracking search and a search heuristic based on the lookahead rule. We have tested the method on symbolic model checking benchmarks against state-of-the-art satisfiability checkers. The experiments indicate that the tableau method provides a promising basis for solving Boolean satisfiability problems. Interesting topics of further research include the development of refined search heuristics that take better into account the circuit structure, intelligent backtracking methods and simplification techniques.

Acknowledgements

The authors wish to thank Patrik Simons for discussing the implementation of the Smodels system and to gratefully acknowledge the financial aid of the Academy of Finland (projects no. 43963 and 47754). Tommi Junttila is grateful for the support from Helsinki Graduate School in Computer Science and Engineering (HeCSE) and Tekniikan Edistämissäätiö ("Foundation of Technology").

References

1. A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In W. R. Cleaveland, editor, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'99)*, volume 1579 of *LNCS*, pages 193–207. Springer, 1999.
2. 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 Design Automation Conference (DAC'99)*, pages 317–320. ACM, 1999.
3. A. Biere, E. Clarke, R. Raimi, and Y. Zhu. Verifying safety properties of a PowerPC microprocessor using symbolic model checking without BDDs. In N. Halbwachs and D. Peled, editors, *Computer Aided Verification: 11th International Conference (CAV'99)*, volume 1633 of *LNCS*, pages 60–71. Springer, 1999.
4. A. Borälv. The industrial success of verification tools based on Stålmarck's method. In *Proceeding of the 9th International Conference on Computer Aided Verification (CAV'97)*, volume 1254 of *LNCS*, pages 7–10, Haifa, Israel, June 1997. Springer.
5. R. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.
6. J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.
7. M. D'Agostino and M. Mondadori. The taming of the cut. *Journal of Logic and Computation*, 4:285–319, 1994.
8. L. Guerra e Silva, L. M. Silveira, and J. Marques-Silva. Algorithms for solving Boolean satisfiability in combinatorial circuits. In *Design, Automation and Test in Europe (DATE'99)*, pages 526–530. IEEE, 1999.
9. T. Junttila. BCSat — a satisfiability checker for Boolean circuits. Available at `http://www.tcs.hut.fi/~tjunttil/bcsat`.
10. H. Kautz, D. McAllester, and B. Selman. Exploiting variable dependency in local search. A draft available at `http://www.cs.cornell.edu/home/selman/papers-ftp/papers.html`, 1997.
11. H. Kautz and B. Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proceedings of the 13th National Conference on Artificial Intelligence*, Portland, Oregon, July 1996.
12. C. Li and Anbulagan. Look-ahead versus look-back for satisfiability problems. In *Principles and Practice of Constraint Programming – CP97*, volume 1330 of *LNCS*, pages 341–355. Springer, 1997.
13. F. Massacci. Simplification — a general constraint propagation technique for propositional and modal tableaux. In H. de Swart, editor, *Proceedings of the International Conference on Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX-98)*, pages 217–231. Springer, May 1998.
14. A. Nerode and R. A. Shore. *Logic for Applications.* Text and Monographs in Computer Science. Springer-Verlag, 1993.
15. I. Niemelä and P. Simons. Efficient implementation of the well-founded and stable model semantics. In M. Maher, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 289–303. The MIT Press, 1996.
16. C. H. Papadimitriou. *Computational Complexity.* Addison-Wesley, 1995.
17. R. Sebastiani. Applying GSAT to non-clausal formulas. *Journal of Artificial Intelligence Research*, 1:309–314, 1994.
18. P. Simons. Towards constraint satisfaction through logic programs and the stable model semantics. Research report A47, Helsinki University of Technology, Helsinki, Finland, August 1997. Available at `http://www.tcs.hut.fi/pub/reports/A47.ps.gz`.
19. H. Zhang. SATO: An efficient propositional prover. In *Automated Deduction – CADE-14*, volume 1249 of *LNCS*, pages 272–275. Springer, 1997.