Propositional Satisfiability: Algorithms and Applications

Anton Belov

Technical Report CSE-2008-06

September 5, 2008

Department of Computer Science and Engineering

4700 Keele Street Toronto, Ontario M3J 1P3 Canada

Propositional Satisfiability: Algorithms and Applications

Anton Belov

Department of Computer Science and Engineering
York University, Toronto, Canada
antonb@cs.yorku.ca

Abstract

In the first part of this paper we survey a number of algorithms for solving the propositional satisfiability problem (SAT). We dedicate a large amount of attention to the *non-clausal SAT algorithms*, that is, the algorithms that work on arbitrary propositional formulas, and to the *circuit SAT algorithms* that work on Boolean circuit representation of formulas. We also discuss some of the non-mainstream clausal SAT algorithms.

The second part of this paper discusses some of the practical applications of SAT, particularly to Bounded Model Checking and to Satisfiability Modulo Theories.

Contents

Chapter 1

Introduction

In this paper we discuss the algorithms for solving the *Propositional Satisfiability Problem (SAT)* and the applications of SAT in practice. SAT is the problem of determining whether there exists a truth-value assignment to a variables of a given propositional formula under which the formula evaluates to 1. Despite the fact that SAT is $\mathcal{NP}$-complete [Cook, 1971], algorithms that work well on a large variety of practical SAT instances have been developed. The first part of this paper is devoted to a detailed presentation of some of these algorithms. The majority of the publications that survey SAT solving algorithms [Lynce and Marques-Silva, 2003, Mitchell, 2005, Gomes et al., 2007] focus on algorithms that represent propositional formulas in Conjunctive Normal Form (CNF)[1] – we call such algorithms *clausal SAT algorithms*. In this paper we take a broader view, and dedicate a large amount of attention to the *non-clausal SAT algorithms*, that is, the algorithms that work on arbitrary propositional formulas, and to the *circuit SAT algorithms* that work on Boolean circuit representation of formulas. We also discuss some of the non-mainstream clausal SAT algorithms, such as those in Section 2.2.2 and 2.3.2.

By the virtue of being an $\mathcal{NP}$-complete problem, SAT attracts a lot of attention from the Theoretical Computer Science community. Many publications have been devoted to the development of the non-trivial upper bounds on the time complexity of solving SAT, and to the investigation of the structure of the space of satisfying truth-value assignments of propositional formulas. To survey a current state-of-the-art of the theory of SAT would require a review on its own, and so we do not attempt it here. A good starting point for such review would be [Dantsin et al., 2001], [Kullmann,

[1] Conjunctive Normal Form is defined on page 4 of this paper.

2000], and [Istrate, 2007].

Instead, in the second part of this paper we focus on the practical applications of SAT. The popularity of SAT in applications (as opposed to other $\mathcal{NP}$-complete problems) can perhaps be explained by the fact that propositional logic is a very convenient formalism for representation of a wide variety of problems. Additionally, algorithms for solving SAT were available already in the 1960's [Davis and Putnam, 1960, Davis et al., 1962], and so a translation into SAT was a natural choice for many applications. The applications, in turn, drove the development of the ever more efficient SAT solving algorithms, resulting in the current implementations that in some cases can handle instances with million variables and ten million clauses.

Before we proceed, we would like to overview the notation and some of the definitions used in this paper. We assume that the reader is familiar with the basic concepts of propositional and first order logics.

Propositional Logic

Propositional formulas will be denoted by the small Greek letters α, β, etc. while small Latin letters p, q, etc. will be used to denote propositional variables. We use the symbols $\neg$, $\wedge$, $\vee$, $\leftrightarrow$, $\rightarrow$ for Boolean operators. The constant symbols true and false will be written as T and F.

Given a propositional formula α, by $Vars(\alpha)$ we denote the set of all propositional variables that occur in α. We will write $\alpha(p)$ to emphasize that $p \in Vars(\alpha)$. The countable set of all propositional variables is denoted by $Vars$. A *truth-value assignment* is a partial function $h : Vars \mapsto \{0, 1\}$, with 0 and 1 being the designated truth-values. If, for a given formula α, a truth-value assignment h is defined for all variables in $Vars(\alpha)$, then h is called a *complete* truth-value assignment for α, otherwise it is *partial*. The variable/truth-value pairs in h will be written as $v \mapsto \nu$.

If β is a subformula of α, and γ is an arbitrary propositional formula, then by $\alpha(\beta/\gamma)$ we will denote the propositional formula obtained from α by the simultaneous replacement of *all* occurrences of β with γ. This operation is called *substitution*. Typically, we only substitute variables by the logical constants T and F, as in $\alpha(p/T)$. Sometimes we will abuse this notation and write $\alpha(p/h(p))$ to indicate that p is replaced with the logical constant which corresponds to the truth-value $h(p)$ (i.e. T if $h(p) = 1$, and F otherwise).

In the descriptions of the algorithms in this paper we will often refer to the procedure SIMPLIFY(), which takes two parameters: a propositional formula α, and a truth-value assignment h. The intended functionality of

this procedure is as follows: for each $p \in Vars(\alpha)$, such that $h(p)$ is defined, apply the substitution $\alpha(p/h(p))$ and simplify the resulting formula by the repeated application of logical simplification rules (e.g. replace $T \wedge p$ with p). The input formula α is assumed to be passed "by-reference" to SIMPLIFY().

A *literal* is a propositional variable or its negation. A *clause* is a disjunction of literals. A *c-clause* is a conjunction of literals. A propositional formula is said to be in *Conjunctive Normal Form (CNF)* if it is a conjunction of clauses, in *Disjunctive Normal Form (DNF)* if it is a disjunction of c-clauses, and in *Negation Norm Form (NNF)* if the only logical connectives that occur in the formula are the negation, conjunction and disjunction, and the negation connective applies to variables only. Every propositional formula has an equivalent formula in CNF, in DNF, and in NNF. By a *k-CNF formula* we refer to a CNF formula in which each clause has exactly k literals. By k-SAT we refer to the problem of determining the satisfiability of k-CNF formulas.

Boolean Circuits

A *Boolean circuit*, or, simply a *circuit* is a directed acyclic graph (DAG) (V, E), with the set of all 0-indegree nodes $I \subset V$ being the set of *input nodes* (or, *inputs*), and the set of all 0-outdegree nodes $O \subset V$ being the set of *output nodes* (or, *outputs*). In this paper we assume $|O| = 1$. Each node $v \in V \setminus I$ is associated with a Boolean function f_v which has the same number of arguments as the indegree of v. This function computes the output value of the node v, given the output values of its predecessors. The *fanin* of a node $v \in V$ is the set of its predecessors, i.e. the set $\{v' | (v', v) \in E\}$. Similarly, the *fanout* of v is the set of all of its successors.

A final remark: in this paper we distinguish between a SAT algorithm and SAT solver, the latter being a computer program that implements the former.

The rest of this paper is organized as follows. In Chapter 2 we discuss clausal, non-clausal and circuit SAT algorithms, in this order. In Chapter 3 we describe some of the applications of SAT – applications to Bounded Model Checking and Satisfiability Modulo Theories are covered in detail. We conclude the paper in Chapter 4 with some ideas for our future research.

Chapter 2

SAT Algorithms

2.1 Introduction

SAT solving algorithms presented in this chapter are classified according to their completeness, and according to the underlying formula representation.

Complete algorithms for SAT are those that are guaranteed to terminate on any input formula within bounded time. If the formula is satisfiable, complete algorithms output the satisfying assignment. If the formula is not satisfiable, most of the complete algorithms just output "no", although some of the more recent implementations also produce a *trace*, from which a resolution refutation of the input formula can be extracted. *Incomplete algorithms*, on the other hand, are not guaranteed to terminate, although when they do, the answer will be correct. Practical incomplete SAT algorithms are aimed only at finding satisfying assignments – if the algorithm terminates, the formula is satisfiable. In general, this type of algorithms can not establish unsatisfiability of the formula. A number of incomplete algorithms aimed at proving unsatisfiability have been recently developed (we mention some in Section 2.3), however these are not yet practical.

Although the vast majority of current SAT solving algorithms rely on a CNF representation, in the past decade algorithms that work directly on arbitrary propositional formulas and on boolean circuits have begun to gain strength. The use of CNF for satisfiability solving was suggested in [Davis and Putnam, 1960] by the authors of what is now considered the first SAT solving algorithm DP[1]. Although simplicity of CNF has its clear advantages, it has one significant drawback – conversion to normal form

[1]It is a little known fact that this algorithm had been discovered some 50 years earlier by L.Löwenheim – see [Chvátal and Szemerédi, 1988] for the references to his work.

makes it difficult to recover the structure of the original problem. As a simple example consider the formula

$$(p \leftrightarrow q) \wedge (p \leftrightarrow \neg q),$$

which is *obviously* unsatisfiable. The equivalent CNF formula

$$(\neg p \vee q) \wedge (p \vee \neg q) \wedge (\neg p \vee \neg q) \wedge (p \vee q),$$

is, of course, also unsatisfiable, however quite a bit more reasoning is required to arrive to this conclusion. This, perhaps simple-minded, example is given here only to demonstrate that the structure of the original problem can, and should, be used to speed-up SAT solving algorithms. Thus, although non-clausal and circuit SAT algorithms are not in the mainstream of SAT research, in this chapter we devote a significant amount of attention to these types of algorithms.

Before we proceed, a few words with regards to the performance evaluation of SAT algorithms. As most of the practical SAT algorithms are extremely complex, they largely resist theoretical analysis, and so empirical studies are often the only way to compare and assess their performance. To this extent, for the past decade the SAT community organizes an annual SAT competition [2], in which a large number of SAT solvers (30 in 2007) are compared based on their ability to solve a wide variety of benchmark problems. Solvers are evaluated both on the number of solved problems, and on the CPU time required to solve each problem. The problems in the competition are split into three categories:

- *Random problems* – these are randomly generated instances of k-SAT, and other restricted classes of SAT. The most common of these is the uniform-random k-SAT instances, in which clauses are obtained by drawing k variables out of n uniformly at random, and negating each with probability 1/2.

- *Crafted problems* – these are encodings of various problems from the complexity class $\mathcal{NP}$ into SAT. Examples include encodings of the graph coloring problem, the quazigroup completion problem, the minimal disagreement parity problem, factorization, and many others.

- *Industrial problems* – these result from translation of various hardware and software verification problems to SAT. Many of these benchmarks come directly from the industry.

[2] SAT competition website is at `http://www.satcompetition.org`

Random problems are of great interest both because they tend to be very difficult for a wide range of SAT algorithms, and because they are sometimes amendable to analytical treatment. Industrial problems are the closest to the most practical applications of SAT and possess a lot of structure (symmetries, repeated sub-problems, variable dependencies, etc.). And crafted instances bridge the gap between the two other classes by combining structure with randomness. Instances in each category are further divided into satisfiable and unsatisfiable.

The rest of this chapter is organized as follows. In Section 2.2 and 2.3 we discuss CNF-based algorithms, complete and incomplete. Section 2.4 and 2.5 are devoted to non-clausal SAT algorithms, and Section 2.6 and 2.7 deal with circuit based algorithms. We conclude this chapter in Section 2.8 with the discussion of the relative strengths and weaknesses of the various classes of SAT algorithms.

2.2 Complete Clausal Algorithms

The majority of complete SAT algorithms for formulas in CNF are based on the David-Putnam-Logemann-Loveland (DPLL) search method [Davis and Putnam, 1960, Davis et al., 1962]. DPLL is a recursive backtracking search procedure with various optimizations specialized to the CNF representation. The search for a satisfying truth-value assignment is performed using the recursive application of the *splitting rule* [Davis et al., 1962]:

> Let the given formula α be put in the form $(\alpha_1 \vee p) \wedge (\alpha_2 \vee \neg p) \wedge \alpha_3$, where α_1, α_2, and α_3 are free of p. Then α is inconsistent if and only if $\alpha_1 \wedge \alpha_3$ and $\alpha_2 \wedge \alpha_3$ are both inconsistent[3].

Thus, to check the satisfiability of a propositional formula α, the algorithm recursively checks the satisfiability of simplified formulas $\alpha(p/T)$ and $\alpha(p/F)$. If both formulas are unsatisfiable, then so is α. Otherwise, the logical constant substituted for p in the satisfiable formula defines the truth-value of p in the satisfying truth-value assignment for α.

The DPLL algorithm originally presented in [Davis et al., 1962] implemented two CNF-specific optimizations: the *unit propagation* procedure, and the *pure literal* rule.

The unit propagation procedure is based on the fact that in order to satisfy a CNF formula, all clauses in the formula must evaluate to 1. Hence, every clause that contains exactly one literal $l(p)$ (so called *unit clause*)

[3] We have adjusted the notation in this quote to match ours.

forces the truth-value of p to be such that $l(p)$ evaluates to 1. Thus, given a CNF formula $\alpha = C_1 \wedge \cdots \wedge C_k$, and a truth-value assignment h, for each unit clause $C_i = l(p)$ in α, unit propagation extends h in such a way that $h(l(p)) = 1$, and simplifies α. If the simplified formula contains unit clauses, the procedure extends h again, and continues until no unit clauses are left. At this point one of the following situations is possible:

- α is simplified to the logical constant F – this means that α is not satisfiable.
- α is simplified to the logical constant T – in this case α is satisfiable, and h is a (partial) assignment that satisfies α.
- α is not a logical constant and cannot be simplified further – in this case h is a partial assignment which is *implied* by α.

Algorithm 1 contains the pseudocode of the procedure.

Algorithm 1 UnitPropagation([*in, out*] α, [*in, out*] h)

Input: α – formula in CNF; h – truth-value assignment (partial).
Output: α – simplified; h – extended to contain implied assignments.

```
1: Simplify(α, h)
2: while α contains unit clause l(p) do
3:     ν ← if l(p) = p then 1 else 0
4:     Simplify(α, {p ↦ ν})
5: end while
```

The second optimization technique, the pure literal rule, is based on the fact that if α contains a variable that always appears negated (resp. always appears unnegated), this variable can be assigned to 0 (resp. 1), and α can be simplified to obtain an equisatisfiable formula α'. This rule, however, has been abandoned shortly after its introduction due to the empirical observation that the benefits of applying the rule do not outweigh its computational cost. Algorithm 2 demonstrates the pseudocode of the "standard" DPLL procedure.

We take this opportunity to introduce some of the terminology used in the SAT community. The variable p selected on line 7 is called the *decision variable*, or the *decision literal* when combined with the selected truth-value. The function Decide(), responsible for the selection of the decision literal, implements some sort of *decision heuristic*, which could be as simple as selecting a random unassigned literal (as it was originally done in [Davis

Algorithm 2 DPLL([*in*] α, [*in, out*] h)

Input: α – formula in CNF; h – truth-value assignment, initially $\emptyset$.
Output: SAT and h satisfying assignment exists; UNSAT – otherwise.

```
1: UnitPropagation(α, h)
2: if α = T then
3:     return SAT
4: else if α = F then
5:     return UNSAT
6: end if
7: ⟨p, ν⟩ ← Decide(α, h)
8: h′ ← h ∪ {p ↦ ν}
9: if DPLL(α, h′) = SAT then
10:    h ← h′
11:    return SAT
12: end if
13: h′ ← h ∪ {p ↦ ¬ν}
14: if DPLL(α, h′) = SAT then
15:    h ← h′
16:    return SAT
17: end if
18: return UNSAT
```

et al., 1962]), or as complex as look-ahead based selection which we will discuss in Section 2.2.2. The assignment of a truth-value to p on lines 8 and 13 is called a *decision assignment*. The depth of the recursion stack at the time of decision assignment is called the *decision level* – the very first decision assignment is considered to be made on decision level 1. Variables assigned on line 1 by the unit propagation procedure are called *implied variables*, or *implied literals* when the truth-value is taken into account. The decision level of the implied literals is considered to be the depth of the recursion stack at the time of assignment minus one – thus, literals assigned before any decisions have been made have decision level 0. When unit propagation determines that the current formula is unsatisfiable, we say that there is a *conflict* – the level of the most recent variable assignment at the time of conflict is the *conflict level*.

Actual implementations of the DPLL algorithm and the unit propagation procedure in SAT solvers differ from the descriptions that we gave earlier. In a typical DPLL-based SAT solver, clauses are kept in the *clause database* as arrays of literals – when a literal is assigned a truth-value, contrary to our earlier description neither clauses nor the literal itself are removed from the database. Instead, the truth-value of the literal is updated inside the clauses that contain it, and clauses are marked as either satisfied or *unresolved*. A clause is unresolved if it has at least one unassigned literal, and all of its assigned literals are false. When an unresolved clause has exactly one unassigned literal, it triggers the unit propagation procedure.

Modern DPLL-based SAT algorithms typically fall into one of two categories: *conflict-driven* algorithms, or *look-ahead* algorithms. Conflict-driven algorithms rely on the fact that conflicts during the DPLL search are caused by certain combinations of variable assignments. By extracting and recording such "bad" combinations, the algorithm can avoid exploring parts of the search space that are known to lead to conflicts. Look-ahead SAT algorithms take a different approach – instead of recording bad variable assignments at the time of conflict, these algorithms try to avoid making bad assignments in the first place. Look-ahead algorithms achieve this by focusing most of the computational effort on the selection of decision variables that lead to the largest reduction of the search space. We now will describe the two categories of algorithms, and will defer the discussion of their comparative strengths and weaknesses to Section 2.8.

2.2.1 Conflict-Driven SAT Algorithms

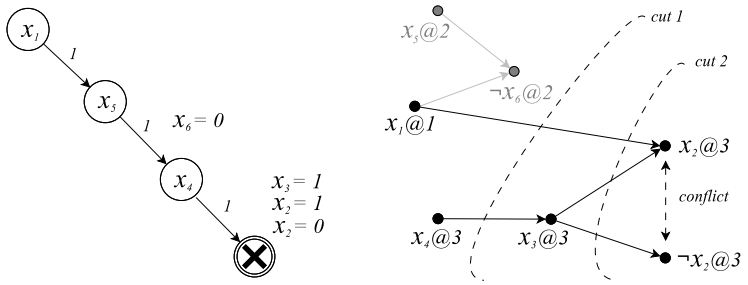
Conflict-driven SAT solvers record variable assignments performed during the search in a datastructure called the *implication graph*. The implication graph is a DAG in which each vertex is associated with a variable assignment. Each assignment is represented as a literal: literal p (resp. $\neg p$) denotes the assignment $p \mapsto 1$ (resp. $p \mapsto 0$). Vertices with in-degree zero correspond to decision assignments. Implied assignments are represented by vertices with incident edges that capture the reasons for the assignments: if l is an assignment made as a result of unit propagation through the clause $l_1 \vee ... \vee l_k \vee l$, then the vertex that corresponds to l will have incoming edges from exactly the vertices that correspond to $\neg l_1, \ldots, \neg l_k$. Construction of the implication graph stops when a variable and its negation are inserted into the graph – this corresponds to a conflict in the DPLL search. The last inserted variable is called a *conflict variable*. To analyze the reasons for the conflict, we only need to consider the subgraph of the implication graph which contains the two conflicting assignments and the vertices that are predecessors of these assignments. Such a subgraph is called a *conflict graph* – the rest of the implication graph is irrelevant to the analysis of the conflict.

As an example, consider a conjunction α of the following set of clauses:

$$
\begin{aligned}
c_1 &= \neg x_1 \vee x_2 \vee \neg x_3 & \qquad c_4 &= \neg x_1 \vee \neg x_5 \vee \neg x_6 \\
c_2 &= \neg x_2 \vee \neg x_3 & \qquad c_5 &= x_4 \vee x_7 \\
c_3 &= x_3 \vee \neg x_4 & \qquad c_6 &= x_4 \vee \neg x_7
\end{aligned}
$$

Assume that the search assigns $x_1 = 1$ at decision level 1, and $x_5 = 1$ at decision level 2. At this point, clause c_4 becomes unit, and x_6 is forced to 0. Next, the algorithm assigns $x_4 = 1$ (decision level 3), at which point clause c_3 becomes unit forcing $x_3 = 1$, which in turn forces $x_2 = 0$ via clause c_2, and $x_2 = 1$ via clause c_1, and so there is a conflict at decision level 3. Figure 2.1(a) depicts the search process, and Figure 2.1(b) shows the corresponding implication graph – note that for clarity we have marked each assignment with the decision level in which the assignment has been made. The grayed out nodes and edges in the implication graph are those that are not in the conflict graph.

Let v be a conflict variable in the conflict graph for CNF formula α. Pick any cut of the graph that has all decision variables on one side (this is the *reason side*), and the conflict literals v and $\neg v$ on the other side (this is the *conflict side*). Let $L = \{l_1, \ldots, l_k\}$ be the set of literals on the reason side



(a) Search tree (b) Implication graph

Figure 2.1: The search tree and the implication graph at the time of first conflict.

that have at least one edge going into the conflict side. It is not difficult to see, that the construction of the conflict graph implies

$$\alpha \models (l_1 \wedge \cdots \wedge l_k \rightarrow v \wedge \neg v). \tag{2.1}$$

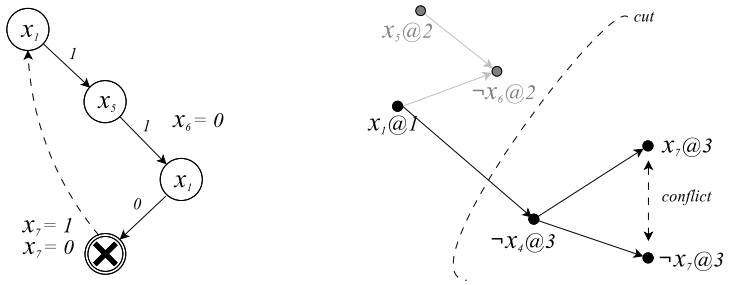
In other words, the assignments in L constitute the *cause* of the conflict, and to avoid getting the same conflict in the future, we can make sure that this combination of assignments will never happen again by conjoining a clause $(\neg l_1 \vee \cdots \vee \neg l_k)$ with α. This clause is called a *conflict clause* (associated with a particular cut). Note, that (2.1) implies that

$$\alpha \models (\neg l_1 \vee \cdots \vee \neg l_k),$$

and so $\alpha \in SAT$ if and only if $\alpha \wedge (\neg l_1 \vee \cdots \vee \neg l_k) \in SAT$.

In our example the conflict clause associated with Cut 1 in Figure 2.1(b) is $cc_1 = \neg x_1 \vee \neg x_4$, and the conflict clause associated with Cut 2 is $cc_2 = \neg x_1 \vee \neg x_3$. Both clauses could be useful in the event the search process backtracks to level 2 and explores the search subtree that corresponds to the assignment $x_5 = 0$.

As we shall see now, however, *non-chronological backtracking* will force the algorithm to skip the subtree $x_5 = 0$ all together. Since the clause $cc_1 = \neg x_1 \vee \neg x_4$ is now added to the formula, the value of the last decision variable x_4 is automatically forced to 0, and clauses c_5, c_6 cause a conflict on variable x_7. The implication graph at this point is shown in Figure 2.2(b). The cut in the graph produces a conflict clause $cc_3 = \neg x_1$. Note that cc_3



(a) Search tree (b) Implication graph

Figure 2.2: The search tree and the implication graph at the time of second conflict.

says that the algorithm should give up on the subtree $x_1 = 1$ – instead of backtracking to the previous level 2, as "standard" DPLL would do, the algorithm can backtrack directly to level 1. The clause cc_3 guarantees that such, non-chronological, backtracks will not miss any satisfying assignments.

Although it may not be apparent from our simple example, the cumulative effect of the addition of conflict clauses and non-chronological backtracking can be profound. Experimental evaluation of the first conflict-driven SAT solver `GRASP` [Marques-Silva and Sakallah, 1996] showed 2 to 5 orders of magnitude improvement in CPU time over the leading at a time complete SAT solvers. Similar results were obtained shortly after that by another conflict-driven SAT solver `rel-sat` [Bayardo and Schrag, 1997]. Six years later, the power of *clause learning* (an umbrella term for conflict analysis and conflict clause extraction and maintenance techniques) has been justified theoretically: in [Beame et al., 2003] the authors exposed a family of unsatisfiable formulas on which the search tree of the "standard" DPLL is exponentially larger than that of DPLL augmented with clause learning.

The performance of conflict-driven SAT solvers was pushed further with the introduction of the *watched literals* technique and the *Variable State Independent Decaying Sum (VSIDS)* variable selection heuristic in SAT solver `zChaff` [Moskewicz et al., 2001]. We briefly describe these below.

The watched literals technique was designed to improve the efficiency of the unit propagation procedure, which at the time of its introduction became a bottleneck of conflict-driven SAT solvers. The main idea of the technique is as follows. When a literal l is assigned 0, a traditional DPLL

implementation visits every clause c which contains l and checks if c is satisfied. If this is the case, no further action is required. Otherwise, a counter of false literals associated with c is incremented. When this counter reaches $|c| - 1$, c triggers unit propagation on the last unassigned literal. The trick introduced in `zChaff` was to maintain pointers to two unassigned literals in each clause – these are the *watched* literals. Each literal l maintains a list of clauses in which it is watched. When l is assigned 0, only the clauses from this list are visited, and each such clause c is checked whether it contains an unassigned, non-watched literal l'. If it does, l' becomes watched for c. Otherwise, c contains exactly $|c| - 1$ literals assigned to 0, and so the second watched literal of c is assigned to 1 by unit propagation. Thus, the watched literals technique significantly reduces the overhead associated with variable assignments. Furthermore, since during backtracking the literals in a clause are un-assigned in the reverse order from which they were assigned, there is no need to change the pointers to watched literals, and the un-assignments can be performed in constant time.

At the time of the introduction of `zChaff` most of the successful variable selection heuristics relied on information about the current state of the search: the number of unresolved clauses for each literal, the count of unassigned literals in unresolved clauses for each literal, etc[4]. The problem with this type of state-dependent heuristics is the computational overhead involved in maintaining and updating various counters needed to implement them. The decision heuristic introduced in `zChaff`, VSIDS, was designed to address this problem: the heuristic selects literals based on a score which is updated only for literals that appear in a conflict clause when it is added to the clause database. Periodically all scores are divided by a constant. Literals with a high score tend to be those that appear in most recent conflict clauses – in other words literals are selected based on their *activity*, as recent conflict clauses correspond to the part of the search space currently explored by the algorithm. Besides the fact that VSIDS has very a low computational overhead, the resulting activity-based search strategy turned out to be very successful, particularly on industrial instances. The authors of [Moskewicz et al., 2001] reported an order of magnitude improvement in CPU time as a result of the introduction of the new heuristic.

Modern conflict-driven DPLL solvers, such as `minisat` [Eén and Sörensson, 2004, 2005], employ all of the techniques outlined above (clause learning, non-chronological backtracking, watched literals, activity based decision heuris-

[4] [Marques-Silva and Sakallah, 1999] gives a good overview of the state-of-the-art in SAT decision heuristics circa 1999.

tics), as well as other important additions:

- The *FirstUIP* learning scheme introduced in [Marques-Silva and Sakallah, 1996, Bayardo and Schrag, 1997] allows to select empirically good cuts in the conflict graph.

- *Conflict clause minimization* [Eén and Sörensson, 2005] attempts reducing the size of conflict clauses: shorter conflict clauses block larger parts of the search space and are faster for unit propagation.

- *Randomized restarts* introduced in [Gomes et al., 1998] and further developed in [Baptista and Marques-Silva, 2000] force DPLL to restart the search process after a certain number of backtracks, while keeping the conflict clauses accumulated in the previous runs. In many cases such restarts significantly improve performance in terms of the CPU time per instance.

To conclude this section, we would like to refer the interested reader to an excellent review of the techniques applied in the state-of-the-art conflict-driven SAT algorithms in [Gomes et al., 2007].

2.2.2 Look-ahead SAT Algorithms

Prior to the introduction of activity-based decision heuristics in [Moskewicz et al., 2001], the conventional wisdom in SAT community was that decision heuristics should try to pick variable assignments that produce the largest reduction in the formula after the application of unit propagation. Such assignments are called *effective assignments*. To find effective assignments, decision heuristics typically targeted variables with a large number of occurrences in short unassigned clauses. For example, a popular at a time heuristic MOM's [Pretolani, 1993] selected variable p that maximizes the function

$$[f^*(p) + f^*(\neg p)] \cdot 2^k + f^*(p) \cdot f^*(\neg p), \tag{2.2}$$

with $f^*(l)$ being the number of occurrences of literal l in the smallest unassigned clauses, and k a constant. The second term in (2.2) is designed to favor variables that occur often in both polarities to improve chances of constructing a short and balanced search tree.

One can view such heuristics as using an approximation of the relative amount of formula reduction as a guideline for selecting decision variables. However, instead of approximating this amount one can calculate it exactly

by performing a *look-ahead* – that is, actually assigning a truth-value to a variable, performing unit propagation, and measuring the reduction. The reduction for variable p is typically expressed as the product of the numbers of clauses reduced as a result of assigning p to 0 and to 1. The DPLL procedure which uses look-ahead to guide decision assignments was first proposed in [Freeman, 1995] – SAT algorithms based on this idea are called look-ahead algorithms.

Besides enabling selection of effective assignments, the look-ahead procedure allows to detect and assign forced variables: for example, if a look-ahead on assignment $p = 1$ results in conflict (p is called a *failed literal* in this case), p is forced to be assigned to 0, and the formula can be simplified. Furthermore, if a look-ahead on assignment $p = 1$ results for example in assignment $q = 1$, a clause $\neg p \vee q$ can be added to the original formula – such clauses are called *local learned clauses*. The idea can be pushed even further by recording the equivalences between variables when both $\neg p \vee q$ and $p \vee \neg q$ are derived as local learned clauses (see, for example, [Li, 2003] and [Heule et al., 2004]).

Despite the high computational cost associated with the look-ahead procedure, look-ahead SAT solvers are extremely efficient, particularly on random and crafted instances. One one of the earliest look-ahead SAT solvers `satz` [Li and Anbulagan, 1997a] outperformed the best at a time DPLL-based solvers (including the conflict-driven `GRASP` and `rel-sat`) by an order of magnitude on hard random 3-SAT instances. Today, the state-of-the-art look-ahead solver `march_ks` [Heule and van Maaren, 2007] is the best performing SAT solver on random unsatisfiable and crafted satisfiable instances. In `march_ks` the basic look-ahead algorithm has been enhanced with a number of important additions which we mention below. [Heule and van Maaren, 2006] and [Heule and van Maaren, 2007] describe all relevant details.

- *Pre-selection heuristics* introduced in [Li and Anbulagan, 1997b] are responsible for selecting a subset of currently unassigned variables on which to perform look-ahead, thus avoiding the cost associated with look-ahead on all unassigned variables.

- *Double look-ahead* was suggested in [Li and Anbulagan, 1997a]. It was observed that when look-ahead produces a large number of binary clauses, the formula is often unsatisfiable. The double look-ahead procedure attempts to detect unsatisfiability by performing additional look-aheads.

- *Direction heuristics* [Heule and van Maaren, 2006] attempt to choose which truth-value of a decision variable to assign first, based on results of look-ahead. This improves performance on satisfiable instances.

We will not discuss look-ahead SAT algorithms further, and instead would like to refer the interested reader to [Heule, 2008] for an excellent, detailed presentation of this class of algorithms.

2.3 Incomplete Clausal Algorithms

Most of the incomplete algorithms for SAT are based on some variant of *stochastic local search (SLS)* – see Algorithm 3. Given a propositional formula α, the search starts by generating a random truth-value assignment h to $Vars(\alpha)$. If $h(\alpha) = 1$, the search terminates. Otherwise a variable $p \in Vars(\alpha)$ is selected, and its truth-value is *flipped* – that is, it is assigned $1 - h(p)$. The process is repeated until either a truth-value assignment that satisfies α is found, or a maximum number of iterations MAX_FLIPS (called, *cutoff*) is reached, in which case the satisfiability status of the formula is undecided. The selection of the variable to flip is usually a two-step process: in the first step, a small set of candidate variables, *candidate list*, is selected from $Vars(\alpha)$ based on a certain candidate list generation strategy (implemented by the function GET_CANDLIST() in Algorithm 3); in the second step, a variable to flip is selected from the candidate list using the *variable selection heuristic* (implemented by function SELECT_VAR()).

Typically, the variable selection heuristic uses some kind of an *objective function* which maps each truth-value assignment to some numeric value. The only requirement to the objective function is that it should take a designated value (e.g. 0) when the assignment is satisfying. In case of CNF, for example, the number of currently unsatisfied clauses is a frequent choice. We will discuss more examples in Section 2.3.1.

Besides SLS, two other successful incomplete algorithms for SAT are known: the *unit propagation based local search* [Hirsch and Kojevnikov, 2005] which uses the unit propagation procedure (Algorithm 1, Section 2.2) to make assignments during the local search, and the *survey propagation* algorithm [Mezard et al., 2002] for hard random k-CNF formulas developed based on insights from statistical physics. We will discuss the former in detail, and refer the interested reader to [Braunstein et al., 2005] for a description of the latter.

Finally, in recent years, incomplete algorithms for *unsatisfiability* started to appear [Prestwich and Lynce, 2006, Audemard and Simon, 2007]. These

Algorithm 3 SLS([*in*] α, [*out*] h)

Input: α – propositional formula;
Output: SAT and h if satisfying assignment is found; UNDECIDED – otherwise.

```
1: h ← random truth-value assignment to Vars(α)
2: flips ← 0
3: while flips < MAX_FLIPS do
4:     if h(α) = 1 then
5:         return SAT
6:     end if
7:     candlist ← GET_CANDLIST(α, h)
8:     p ← SELECT_VAR(α, h, candlist)
9:     h(p) ← 1 − h(p)                          ▷ the flip
10:    flips ← flips + 1
11: end while
12: return UNDECIDED
```

algorithms use local search to search through the space of incomplete proof graphs (for example, resolution refutations) induced by a given CNF formula. At this point, the algorithms are in the initial stages of development, and do not appear to be of practical value.

2.3.1 Stochastic Local Search Algorithms

In this section we review only some of the CNF SAT algorithms based on SLS. We choose algorithms that, in our opinion, had major impact on the development of SLS-based SAT solvers. We refer the interested reader to [Hoos and Stutzle, 2000] or [Hoos and Stutzle, 2005] for a detailed catalog of various SLS-based methods for SAT.

GSAT, GWSAT and GSAT/Tabu

One of the first SLS algorithms for clausal satisfiability, GSAT, was introduced in [Selman et al., 1992] and, independently, in [Gu, 1992]. The candidate list in GSAT is the set of all variables in the input formula, and the variable selection heuristic always picks the variable whose flip minimizes the number of unsatisfied clauses – if there are several such variables, one selected uniformly at random. Thus, GSAT is a greedy search that uses the number of unsatisfied clauses as an objective function, and, as such, the algorithm is prone to get stuck in local minima (or, local plateaus). To

remedy this problem, the search is restarted from a new random assignment whenever a cutoff value is reached. The search is repeated up to a specified maximum number of *tries* before declaring the formula undecided. At the time of its introduction, GSAT significantly outperformed the best complete algorithms on random, crafted and some industrial instances.

The addition of restarts to GSAT is very significant: without restarts even the arbitrary long runs of the algorithm are not guaranteed to find a solution (assuming it exists, of course). Algorithms with this property are called *essentially incomplete*. On the other hand, if an arbitrary number of restarts is allowed, while the cutoff is fixed, the probability that GSAT finds a solution converges to 1 as run-time approaches infinity – this type of algorithms is called *probabilistically approximately complete (PAC)*. Even though it is not clear whether a PAC algorithm will always perform better than an essentially incomplete algorithm (since the convergence rate for a PAC algorithm can be arbitrary small), in practice it has been often observed that variants of SLS algorithms that are PAC do perform significantly better than variants that are non-PAC [Hoos, 1999, Hoos and Stutzle, 2000].

Another modification that gives the PAC property to GSAT is the addition of *conflict-directed random walk*, suggested in [Selman and Kautz, 1993, Selman et al., 1994]. With probability wp (called *walk probability*) the variable selection heuristic picks a variable that occurs in some unsatisfied clause; with probability $1 - wp$ the heuristic follows the GSAT heuristic. The resulting algorithm GWSAT performed significantly better than GSAT ([Selman et al., 1994]). Since then, random walk became an essential part of all SLS algorithms for SAT.

Another idea that had significant impact on modern SLS solvers is that of preventing the repetition of local moves by recording the *age* of each variable – that is, the number of search steps taken since the variable was last flipped. A variable becomes *tabu*, that is, the algorithm is prohibited from flipping it, if its age is smaller than a certain threshold value tt (called *tabu tenure*). The solver `TWSAT`, introduced in [Mazure et al., 1997], which combined GSAT with tabu-based restrictions (GSAT/Tabu algorithm), outperformed both GSAT and GWSAT on many problems – in fact, it is the best performing variant of GSAT to date. Note, however, that it is not clear whether GSAT/Tabu with fixed cutoff is PAC.

WalkSAT

The WalkSAT algorithm, introduced as WSAT in [Selman and Kautz, 1993], is a subtle but significant modification of the GWSAT algorithm presented

above. First, in WalkSAT the candidate list is selected to be the list of all variables that appear in some *currently unsatisfied clause*. Second, the variable selection heuristic in WalkSAT is guided by the so-called *break-value* of a variable, which is the number of currently satisfied clauses that become unsatisfied ("broken") if the variable is flipped. If there is a variable in the candidate list with break-value of 0 that variable is always selected (*zero-damage step*; ties are broken at random). Otherwise, with probability wp the heuristic selects a random variable from the candidate list (random walk, as in GWSAT), and with probability $1 - wp$ a variable with the smallest break value (this is the *greedy move*, ties are broken at random, again). Algorithm 4 demonstrates the variable selection procedure used in WalkSAT:

Algorithm 4 SELECT_VAR_WALKSAT([*in*] α, [*in*] h, [*in*] *candlist*)

Input: α; h; *candlist*.
Output: v – variable to flip
1: **if** *candlist* contains variable v with break-value 0 **then**
2: **return** v ▷ zero-damage step
3: **end if**
4: **with-probability** wp **do**
5: $v \leftarrow$ random variable from *candlist* ▷ random walk
6: **end with-probability**
7: **with-probability** $1 - wp$ **do**
8: $v \leftarrow$ variable with smallest break-value in *candlist* ▷ greedy move
9: **end with-probability**
10: **return** v

Note that although closely related to GWSAT, the WalkSAT algorithm explores the search space in quite a different manner. On one hand, WalkSAT is greedier than GWSAT in that it applies the random walk step only when there is no variable with zero break-value. Thus, WalkSAT will make improving steps as long as it can. On the other hand, when there is no strictly improving variable, WalkSAT examines a significantly smaller number of candidates than GWSAT, and in this sense WalkSAT behaves less greedy.

Generally WalkSAT outperforms GWSAT in terms of number of flips, but does not always reach the performance of GSAT/Tabu. However, due to the fact that WalkSAT examines only a small number of candidate variables, the average flip speed is significantly higher than in any of the GSAT variants, and WalkSAT typically outperforms these algorithms in terms of CPU time [Hoos and Stutzle, 2000]. Similar to GSAT/Tabu, a WalkSAT

variant in which a variable's age is used to guide the variable selection heuristic, WalkSAT/TABU, has been suggested in [McAllester et al., 1997]. Even though the algorithm is essentially incomplete (as opposed to WalkSAT) [Hoos, 1998], WalkSAT/TABU typically performs significantly better than WalkSAT, particularly on crafted instances.

The walk probability parameter *wp* in WalkSAT (and in GWSAT) has a major impact on the performance of the algorithm [McAllester et al., 1997, Hoos and Stutzle, 2000]. The optimal walk probability varies from instance to instance – sometimes this can be the case even for instances from the same general class. Typically, the walk probability is hand-tuned during preliminary runs of the algorithm, which is both time-consuming, and does not necessarily produce optimal results. To avert this problem, in [Hoos, 2002] the authors proposed an *adaptive noise* mechanism for WalkSAT, which monitors the progress of the algorithm, and gradually decreases the walk probability as long as the algorithm makes progress. The walk probability is increased when the algorithm appears to be stuck in a local plateau. The proposed mechanism was shown to have good performance, in some instances even outperforming a hand-tuned version of WalkSAT. Since then, the adaptive noise mechanism has been introduced in a number of other SLS-based SAT algorithms.

Novelty+

The Novelty+ algorithm [McAllester et al., 1997, Hoos, 1998], is a result of the combination of the candidate list generation strategy and the conflict-directed random walk from WalkSAT, with the idea of a history-based variable selection heuristic, as in GSAT/Tabu. The algorithm has two parameters: the walk probability *wp* (as in WalkSAT), and the *novelty noise p*. As in WalkSAT, the candidate list in Novelty+ is a set of variables from a randomly selected unsatisfied clause. To select the variable to flip, Novelty+ uses the age of the variable (as in GSAT/Tabu), and the GSAT objective function – that is, the number of unsatisfied clauses if the variable is flipped (variable's *score*). The algorithm is presented in Algorithm 5.

For $p > 0$, the age-based restrictions of Novelty+ prevent the algorithm from flipping the same variable over and over again, and at the same time the score-based restrictions allow for a bad flip to be reversed, if no alternative is available. The small amount of random walk prevents the algorithm from getting stuck in unprofitable regions of the search space. Novelty+ combined with an adaptive noise mechanism (AdaptiveNovelty+ [Hoos, 2002]) produced the highest-performing SAT algorithm for random instances in the

Algorithm 5 SELECT_VAR_NOVELTY+([*in*] α, [*in*] h, [*in*] *candlist*)

Input: α; h; *candlist*.
Output: v – variable to flip

```
1: with-probability wp do
2:     v ← random variable from candlist                    ▷ random walk
3:     return v
4: end with-probability
5: with-probability 1 − wp do
6:     v ← variable with the smallest score in candlist
7:     if v is not the youngest in candlist then
8:         return v
9:     else
10:        with-probability p do                            ▷ novelty noise
11:            v ← variable with second smallest score in candlist
12:        end with-probability
13:        return v
14:    end if
15: end with-probability
```

SAT 2004 competition.

Dynamic Local Search

The main idea of *dynamic local search (DLS)* algorithms is to associate a weight with each clause in the CNF formula, and use a weighted objective function to guide the variable selection heuristic. A sum of weights of the currently unsatisfied clauses is an example of such an objective function, commonly used in DLS-based algorithms. Initially, all clause weights are initialized to the same value (1, for example), and as the search progresses, the values of clauses are adjusted – typically, the clauses that are deemed "difficult to satisfy", according to some criterion, have their weights incremented, so that the search process focuses on these clauses. The earliest example of a DLS algorithm for SAT, proposed in [Selman and Kautz, 1993], simply increases the weights of the clauses unsatisfied at the end of each try of GSAT.

Many successful state-of-the-art DLS algorithms for SAT are based on the Exponentiated Subgradient algorithm (ESG) proposed in [Schuurmans et al., 2001]. ESG starts by assigning weight 1 to all of the clauses of the formula. At each step, ESG selects all variables in a random unsatisfied

clause as a candidate list, and chooses a variable whose flip will minimize the total weight of unsatisfied clauses in the formula. When a local minimum is reached (that is, no flip leads to a decrease in the objective function), with a certain probability η the algorithm chooses a random variable from all currently unsatisfied clauses, and continues the local search. However, with probability $1 - \eta$ the algorithm enters the weight update phase, which is performed in two stages. In the *scaling stage* the weights of all clauses are multiplied by a factor which depends on their satisfaction status: α_{sat} for satisfied clauses, α_{unsat} for unsatisfied ones. In the *smoothing stage* all clause weights are adjusted towards their mean using the formula $weight(c) = \rho \cdot weight(c) + (1 - \rho) \cdot \bar{w}$, where $\bar{w}$ is the current mean clause weight, and $0 < \rho < 1$. The local search is then continued. The procedure is repeated until either the satisfying assignment is found, or a maximum number of iterations is reached.

While ESG was superior in terms of number of search steps to the best at the time SLS algorithm Novelty+, the computational cost of weight updates had a negative impact on the runtime of the algorithm. Subsequent work of authors in [Schuurmans et al., 2001] resulted in a significant improvement of ESG. However, it was the Scaling and Probabilistic Smoothing (SAPS) algorithm proposed in [Hutter et al., 2002] that succeeded to outperform Novelty+ on many instances in terms of CPU time.

The most successful SLS-based SAT algorithm known today, gNovelty+ [Pham and Gretton, 2007], the winner of SAT 2007 competition on random satisfiable instances, incorporates ideas from both the Novelty+ and SAPS algorithms.

2.3.2 Unit Propagation Local Search

The unit propagation local search algorithm, UnitWalk, [Hirsch and Kojevnikov, 2005] was developed based on the results of theoretical work in [Paturi et al., 1997] and [Paturi et al., 1998] on weakly exponential worst-case upper bounds for SAT. We will try to give some intuition behind these results – please consult the cited papers for further (very interesting) details.

Consider a CNF formula $\alpha(p_1, \ldots, p_n)$ that has exactly one satisfying assignment h_S. Pick a random truth-value assignment h – if n values in h were guessed correctly (same as in h_S), then we have a satisfying assignment. However, note that not all n values must be guessed. Take $p \in Vars(\alpha)$, assume $h_S(p) = 1$, and let C be a clause in which p is the only true literal under h_S. Such C must exist, as otherwise we could flip p and obtain

another satisfying assignment. Now, if we guess correctly the assignments to all literals in C except p, then the correct assignment to p can be derived using unit propagation. Of course, for this to work we also need to guess in the correct order – all variables in C must be assigned before p. It turns out, that for *any* k-CNF formula, the expected number of variables that need to be guessed correctly, over all possible orderings of variables, is *at most* $(n - n/k)$ – this, in essence, is the *Satisfiability Coding Lemma (SCL)* in [Paturi et al., 1997].

We can now construct ("guestruct" would be a better word) a satisfying assignment h for a k-CNF formula α using the following procedure. Pick a random truth-value assignment h_R, fix an ordering of the variables in h_R, and for each variable p according to this ordering do the following: if α contains a unit clause p (resp. $\neg p$), and does not contain unit clause $\neg p$ (resp. p), then $h(p) = 1$ (resp. $h(p) = 0$), otherwise $h(p) = h_R(p)$. In any case, simplify α to be $\alpha(p/h(p))$. According to the SCL, the truth-value assignment h obtained at the end of this procedure will be satisfying with probability $1/2^{n-n/k}$, and so by repeating this procedure $O(2^{n-n/k})$ times we find the satisfying assignment with a probability arbitrary close to 1.

Motivated by the procedure described in the previous paragraph, the authors in [Hirsch and Kojevnikov, 2005] proposed the algorithm, UnitWalk, outlined in Algorithm 6. The inner loop of the algorithm, called a *period*, is an implementation of the assignment construction procedure, as described above, with two modifications. First, all unit clauses are processed as soon as they appear, and not only the ones that contain the variable assigned according to the current ordering. Second, the truth-value assignment obtained at the end of the period is used as a seed for the next period. Additionally, if at the end of the period no modifications to the initial assignment have been made, a random variable is flipped – although in practice this happens very rarely, this is essential to make the algorithm PAC.

The SAT solver `UnitWalk`, based on the UnitWalk algorithm, includes additional enhancements to the basic procedure such as the combination with SLS and periodic addition of short resolvents, all of which are described in full in [Hirsch and Kojevnikov, 2005]. The solver won the SAT 2003 competition on random satisfiable instances.

2.4 Complete Non-clausal Algorithms

The DPLL procedure presented in Algorithm 2 in Section 2.2 can be generalized in a straightforward manner to work on non-clausal formulas. The

Algorithm 6 UnitWalk([*in*] α, [*out*] h)

Input: α – formula in CNF
Output: SAT and h if satisfying assignment is found; UNDECIDED – otherwise

$h \leftarrow$ random truth-value assignment to $Vars(\alpha)$
$periods \leftarrow 0$
while $periods <$ MAX_PERIODS **do**
    $\pi \leftarrow$ random permutation of $Vars(\alpha)$
    $\alpha' \leftarrow \alpha$
    for all $p \in \pi$ **do**
        while α' contains unit clauses **do** ▷ make all propagations
            $l \leftarrow$ a random unit clause from α';
            $v \leftarrow$ the variable in l
            if α' does not contain unit clause $\neg l$ **then**
                $h(v) \leftarrow$ if $l = v$ then 1 else 0 ▷ unit-directed flip
            end if
            Simplify($\alpha', \{v \mapsto h(v)\}$)
        end while
        if $p \in Vars(\alpha')$ **then** ▷ p has not been eliminated
            Simplify($\alpha', \{p \mapsto h(p)\}$)
        end if
        if $\alpha' = T$ **then**
            return SAT
        end if
    end for
    if no flips in h were made **then**
        flip random variable in h
    end if
    $periods \leftarrow periods + 1$
end while
return UNDECIDED

only part that needs to be modified is the unit propagation procedure which relies on the fact that the input formula is in CNF. Unit propagation can be replaced with the *Boolean Constraint Propagation (BCP)* procedure [McAllester, 1980, 1990]. When a non-clausal formula is viewed as a tree, BCP is simply a repeated application of the set of rules based on the truth-tables of connectives of propositional logic to the nodes of the tree. For example, "if an AND node is assigned 1, assign 1 to all its children", or "if an OR node assigned 1 and all but one of its children are 0, assign 1 to the remaining child". Note that if a CNF formula is viewed as a 2-level tree, the two aforementioned rules define exactly the unit propagation procedure.

The first such generalization of DPLL that we are aware of appeared in [Van Gelder, 1988], where also a non-trivial worst-case running time of the algorithm was developed. Since such a generalized DPLL procedure would also work on circuits, we postpone the discussion of the procedure to Section 2.6, and, instead, in this section focus on algorithms designed specifically for non-clausal formula trees.

One of such algorithms, presented in [Gutiérrez et al., 2002], is based on the special datastructure, Δ-tree [Gutiérrez et al., 2000], designed to represent and support various operations on NNF formulas. It is difficult to say whether the algorithm is competitive with CNF-based SAT algorithms, as the authors experimented with very small formulas, and did not compare their implementation with any CNF-based SAT solver. Nevertheless, the results presented by the authors indicate that the proposed algorithm could be of interest.

Another interesting SAT solving algorithm for non-clausal formulas is based on a first-order proof procedure called *General Matings*. We discuss the algorithm below.

2.4.1 General Matings

General Matings is a first-order proof procedure originally proposed in [Andrews, 1981]. One its distinguishing features is that the propositional fragment is handled directly in non-clausal form. In [Jain et al., 2006] the authors propose a non-clausal SAT procedure based on General Matings.

At the heart of the procedure is a special 2-dimensional format for representation of NNF formulas called *vertical-horizontal path form (vhpform)*. In this form disjuncts are arranged horizontally, so $\alpha \vee \beta$ is represented as $[\alpha \vee \beta]$, while conjuncts are arranged vertically, so $\alpha \wedge \beta$ is represented as

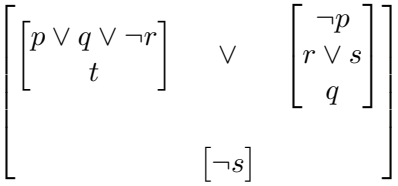


Figure 2.3: The vhpform of NNF formula (2.3).

$\begin{bmatrix}\alpha\\ \beta\end{bmatrix}$. Figure 2.3 demonstrates the vhpform for the formula

$$(((p \vee q \vee \neg r) \wedge t) \vee (\neg p \wedge (r \vee \neg s) \wedge q)) \wedge \neg s. \tag{2.3}$$

Vhpforms can be analyzed in terms of vertical and horizontal paths. A *vertical path* through a vhpform is a sequence of literals formed by choosing one disjunct from each disjunction, and deleting all parts of the vhpform that are not chosen. For example, the set of all vertical paths through the vhpform in Figure 2.3 is $\{\langle p, t, \neg s\rangle, \langle q, t, \neg s\rangle, \langle \neg r, t, \neg s\rangle, \langle \neg p, r, q, \neg s\rangle, \langle \neg p, s, q, \neg s\rangle\}$. Similarly a *horizontal path* through a vhpform is a sequence of literals formed by choosing one conjunct from each conjunction, and deleting all parts of the vhpform that are not chosen. The set of all horizontal paths in Figure 2.3 is $\{\langle p, q, \neg r, \neg p\rangle, \langle p, q, \neg r, r, s\rangle, \langle p, q, \neg r, q\rangle, \langle t, \neg p\rangle, \langle t, r, s\rangle, \langle t, q\rangle, \langle \neg s\rangle\}$.

Let $VP(\alpha)$ and $HP(\alpha)$ be the sets of all vertical and all horizontal paths in a vhpform of NNF formula α. It is not very difficult to see that vertical paths correspond exactly to the c-clauses in a DNF of α, and that, in fact, $\alpha \equiv \bigvee_{\pi \in VP(\alpha)} \bigwedge_{l \in \pi} l$. Similarly, horizontal paths correspond to clauses in a CNF of α, and $\alpha \equiv \bigwedge_{\pi \in HP(\alpha)} \bigvee_{l \in \pi} l$. Therefore,

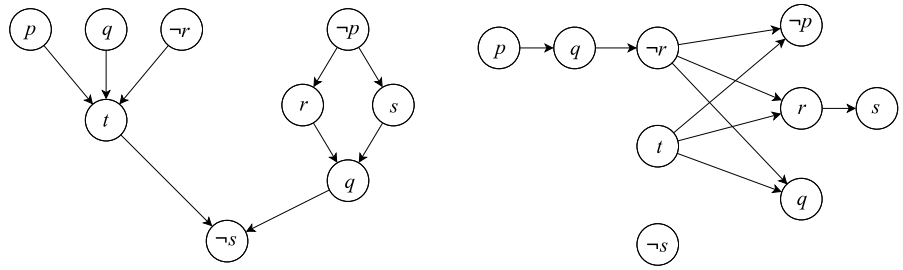
- α is satisfiable if and only if there exists a vertical path in the vhpform of α that does not contain two complimentary literals. This path represents a (partial) satisfying assignment for α.

- α is a tautology if and only if every horizontal path in the vhpform of α contains two complimentary literals.

To determine the satisfiability of an arbitrary propositional formula α, the algorithm in [Jain et al., 2006] transforms α into NNF (when α does not contain equivalence connectives, this can be done at linear cost), and searches the corresponding vhpform for a vertical path that does not contain two complimentary literals. If such path is found, then α is satisfiable, and the path gives a satisfying assignment for α, otherwise α is not satisfiable.

The algorithm is guaranteed to terminate because the number of vertical paths in any vhpform is finite.

To implement the search efficiently the authors construct two directed acyclic graphs - *vgraph* to represent all vertical paths, and *hgraph* to represent all horizontal paths. Figures 2.4(a) and 2.4(b) demonstrate the graphs for our example formula (2.3). Note that the graphs can be easily constructed without explicit construction of the vhpform.



(a) vgraph (b) hgraph

Figure 2.4: The vgraph and hgraph that correspond to vhpform in Figure 2.3.

The vgraph is searched for a *satisfying path* – the path from one of the root nodes to one of the leaf nodes that does not contain complimentary literals. At each step, the algorithm attempts to extend the current partial path $\pi = \{l_1, \ldots, l_k\}$ by adding one of the children l of the node l_k in the vgraph. If $l = \neg l_i$ for some $l_i \in \pi$, the algorithm backtracks. If the hgraph contains a path with exactly the literals $\{\neg l_1, \ldots, \neg l_k, \neg l\}$ the algorithm backtracks as well, because this path represents a falsified clause in the CNF representation of α – i.e. a conflict. If the hgraph contains a path with exactly the literals $\{\neg l_1, \ldots, \neg l_k, \neg l, l'\}$ for some unassigned literal l' , then l' is assigned truth-value 1 – this is the equivalent of unit propagation in clausal solvers. The performance of the algorithm is further enhanced by addition of clause learning and non-chronological backtracking. See [Jain et al., 2006] for further details of the algorithm[5].

The experimental evaluation of the General Matings SAT algorithm performed by the authors, indicates that the algorithm works particularly well

[5]See also SatMate website at `http://www.cs.cmu.edu/ modelcheck/satmate/`, which unpublished an document with detailed description of the solver.

on crafted problems. The performance of the `SatMate` [URL-b] solver based on the algorithm is at least competitive, and in some instances significantly better than the performance of state-of-the-art CNF based conflict-driven SAT solvers. However, the algorithm did not fare well on hardware verification industrial benchmarks.

2.5 Incomplete Non-clausal Algorithms

With the success of SLS-based algorithms in the 1990's it was natural to attempt to generalize this class of algorithms to non-clausal formulas. The first such attempt was published in [Sebastiani, 1994] where the author generalized the GSAT procedure (Section 2.3.1) to formulas in NNF by using a scoring function which, in effect, counted the number of unsatisfied clauses in the equivalent CNF representation of the non-clausal formula. The author did not present the results of experimental evaluation of the algorithm, and we suspect that since the scoring function did not take any advantage of the underlying structure of the NNF formula, the algorithm would not be competitive with CNF-based algorithms.

The first successful non-clausal SLS based algorithm, DAGSAT, was presented in [Kautz et al., 1997]. Although positioned as an algorithm for boolean DAGs (circuits), the algorithm transformed the input DAG into an NNF formula tree before solving it, hence we consider it in this section. The NNF representation used in the algorithm was constructed in such a way that the resulting formula tree had alternating AND and OR levels, starting from the AND at the root. The algorithm was based on WalkSAT (Section 2.3.1) and used the number of false OR nodes in the second level of the tree as the objective function. The candidate list was selected by implicitly constructing a violated "virtual clause" – that is, the clause that would be violated in the equivalent CNF representation of the formula. The SAT solver based on the DAGSAT algorithm significantly outperformed WalkSAT on randomly generated formula trees, however did not do well on circuit verification problems.

The algorithm we discuss here, polSAT, first developed in [Stachniak, 2002], and further improved in [Stachniak and Belov, 2008], is an SLS-based algorithm which uses the concept of *logical polarity* [Stachniak, 1999] to guide the search for satisfying assignments.

2.5.1 Polarity Guided Local Search

Given a propositional formula α in NNF, an occurrence of variable $p \in Vars(\alpha)$ is said to be negative if it is negated, and positive otherwise. A variable $p \in Vars(\alpha)$ is said to have *positive polarity*, if all of its occurrences are positive, and *negative polarity* if all of its occurrences are negative. If p has both positive and negative occurrences, then it is said to have no polarity. A formula α is *polarized* if all of its variables are polarized. Polarized formulas can be seen as generalization of clauses – each non-tautological clause is a polarized formula. Just as with unit propagation for clauses, BCP through a set of polarized formulas can be performed very efficiently [Stachniak, 1999].

When solving satisfiability, the concept of polarity can be useful as well. For example, if $p \in Vars(\alpha)$ is positive, then $\alpha(p) \models \alpha(p/T)$, and so $\alpha \in SAT$ if and only if $\alpha(p/T) \in SAT$. In other words, every polarized variable can be substituted for an appropriate logical constant to obtain an equisatisfiable formula – this is analogous to the pure literal rule for DPLL mentioned in Section 2.2. Furthermore, by considering polarities of variable occurrences only, an objective function for non-clausal local search can be developed as in suggested in [Stachniak, 2002]. This objective function, used in the polSAT algorithm, is defined in terms of the *polarity clash* of formula α with respect to a truth-value assignment h:

Definition 1. *Let α be a formula in NNF and h be a truth-value assignment for α. The* polarity clash *of α with respect to h, $clash(\alpha, h)$, is:*

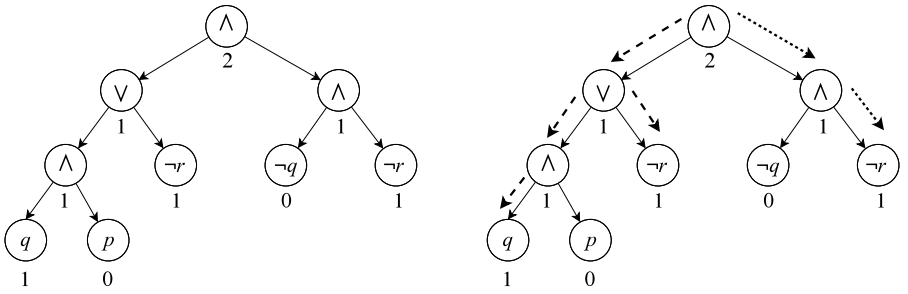
- *if α is a literal, then $clash(\alpha, h) = 1 - h(\alpha)$;*
- *if $\alpha = \beta_1 \vee \cdots \vee \beta_k$, then $clash(\alpha, h) = \min_{1 \leq i \leq k} clash(\beta_i, h)$;*
- *if $\alpha = \beta_1 \wedge \cdots \wedge \beta_k$, then $clash(\alpha, h) = \sum_{1 \leq i \leq k} clash(\beta_i, h)$.*

Intuitively, the clash value of a formula indicates the "amount of work" (in flips) required to make the formula true. It is not difficult to see that $clash(\alpha, h) = 0$ if and only if $h(\alpha) = 1$. As an example, consider the formula

$$\alpha = ((q \wedge p) \vee \neg r) \wedge (\neg q \wedge \neg r) \tag{2.4}$$

depicted in Figure 2.5(a) as a tree. The clash values of subformulas under the truth-value assignment $h = \{p \mapsto 1, q \mapsto 0, r \mapsto 1\}$ are marked under the nodes of the tree, and $clash(\alpha, h) = 2$.

Note that even though in Definition 1 we assume that α is in NNF, the concept of polarity clash can be generalized to arbitrary formulas in



(a) Clash calculation

(b) Candidate list generation

Figure 2.5: Polarity clash calculation and candidate list generation for formula (2.4). Clashes are marked under the nodes. Dashed and dotted paths in (b) show two different ways to collect candidates.

a straightforward manner. Nevertheless, since SAT solvers based on the polSAT algorithm convert their input to NNF for efficiency reasons, in our discussion we assume that non-clausal formulas are always in NNF.

polSAT forms candidate lists by collecting all variables that can be reached from the root of the formula in the following manner:

- at a conjunction with non-zero clash value branch into one random child that has non-zero clash value;
- at a disjunction with non-zero clash value branch into all children.

Figure 2.5(b) shows two possible ways to collect candidates for the formula (2.4). Following the dashed path we obtain $\{q, r\}$, and following the dotted path we obtain $\{r\}$.

Let $cl(\alpha, h)$ be the candidate list for formula α and truth-value assignment h constructed in the above manner. As shown in [Stachniak and Belov, 2008], $cl(\alpha, h)$ has the following nice property: for every satisfying truth-value assignment h_S for α, $cl(\alpha, h)$ contains at least one variable p, such that $h(p) \neq h_S(p)$. In other words, we know that at least one variable in the candidate list must be flipped in order to arrive at a satisfying assignment. To select the variable to flip, polSAT may use any of the clausal SLS variable selection heuristics described in Section 2.3.1 – the only modification required is to use the clash value instead of the number of unsatisfied clauses. For example, the polSAT-N algorithm in [Stachniak and Belov, 2008] uses

the variable selection heuristic of the AdaptiveNovelty+ algorithm [Hoos, 2002]. It is rather interesting that good clausal variable selection heuristics seem to work well in the non-clausal setting as well.

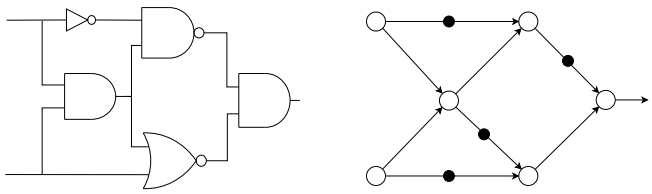
The performance of the SAT solver `polSAT`, based on the polSAT-N algorithm, is competitive with the state-of-the-art CNF-based SLS solvers. In fact, on some classes of random and industrial problems, `polSAT` is faster by 2-3 orders of magnitude (in terms of CPU time). Though for some problems, it seems that the cost of additional computations outweighs the benefits of non-clausal representation.

Before we conclude the discussion of polSAT, we would like to mention another interesting aspect of the candidate lists generated by polSAT. The fact that the candidate list $cl(\alpha, h)$ contains at least one variable that must be flipped in order to satisfy α can be represented by the clause $\bigvee_{p \in cl(\alpha,h)} l(p)$, where $l(p) = \neg p$ if $h(p) = 1$ and $l(p) = p$ if $h(p) = 0$. Such a *learned clause* can easily be shown to be an implicate of α, and in fact, the conjunction of all possible learned clauses is a CNF representation of α. Thus, learned clauses can be used to extract variable dependencies (equivalences, implications, etc.) from α on-the-fly, and, to refine the variable selection heuristic by combining the clausal and non-clausal information. Some of these ideas are explored in [Stachniak and Belov, 2008].

2.6 Complete Algorithms for Circuits

SAT algorithms for circuits were developed in the early 1980s in the context of *Automated Test Pattern Generation (ATPG)*. One of the tasks in ATPG is to find a set of assignments to inputs of a circuit that result in a specified output value of one of the internal nodes. The backtrack search algorithm for this problem proposed in [Goel, 1981] can be seen as a generalization of the DPLL procedure (Section 2.2) which takes into account the structure of the circuit. In the modern literature this procedure is often called *circuit DPLL*, we will use this name as well.

Conceptually, circuit DPLL has one essential modification from the original DPLL – the algorithm is allowed to make decision assignments to the internal nodes of the circuit in addition to circuit inputs. Although this was originally motivated by the nature of the ATPG problem, [Järvisalo et al., 2005] demonstrated a family of circuits on which DPLL restricted to branching on input nodes only must explore an exponentially larger search tree than DPLL allowed to branch on all circuit nodes. Thus, branching on internal nodes is essential for the algorithm's performance. Also, in cir-



(a) Circuit (b) AIG

Figure 2.6: A Boolean circuit, and the corresponding AIG.

cuit DPLL branching is allowed only on values of inputs of the currently assigned but not *justified* nodes. A node is justified when its output value is implied by its inputs. This way the search in circuit DPLL is focused only on parts of the circuit that are known to have impact on the values of currently assigned nodes. A moment of reflection will make one realize that the clausal DPLL procedure acts in a similar manner, when one considers a CNF formula as a two-level circuit (with a top-level AND node, and OR nodes for each of the clauses).

Most of the current implementations of circuit DPLL rely on the simplified circuit representation called *And-Inverter Graph (AIG)* [Kuehlmann and Krohm, 1997]. In this representation circuits are constructed from 2-input AND gates and inverters only. Graphically such circuits are represented as DAGs with internal nodes being AND gates, and inverters marked as dots on the edges. Figure 2.6 shows a simple circuit, and its AIG representation. During construction of AIGs, common sub-circuits can be easily identified [Ganai and Kuehlmann, 2000], and so resulting AIG is typically more compact than the original circuit. See [Bjesse and Boralv, 2004] for discussion of this and other representations of circuits used in the modern EDA tools.

The pseudocode for the DPLL procedure on AIG circuits is shown in Algorithm 7. The procedure is invoked by passing an output node *out*, the desired truth-value ν, and an empty truth-value assignment h. The algorithm maintains a list of nodes that need to become justified – this is the justification queue. When all nodes are justified, the circuit is satisfiable (meaning that there is an assignment to inputs that results in the truth-value ν on the output node *out*). If the algorithm fails to justify the node *out*, the circuit is unsatisfiable.

Algorithm 7 CIRCUIT_DPLL([*in*] n, [*in*] ν, [*in, out*] h)

Input: n – node; ν – truth-value; h – truth-value assignment, initially $\emptyset$.
Output: SAT and h if satisfying assignment exists; UNSAT – otherwise;

1: **if** ! IMPLY(n, ν, h) **then**
2: **return** UNSAT
3: **end if**
4: **if** $justification\_queue.empty()$ **then**
5: **return** SAT
6: **end if**
7: $n \leftarrow justification\_queue.get\_node()$
8: $h' \leftarrow h$ ▷ backup assignment
9: **if** CIRCUIT_DPLL$(n.left, 0, h)$ **then** ▷ or 1 if $n.left$ is negated
10: **return** SAT
11: **end if**
12: $h \leftarrow h'$ ▷ undo assignments
13: **if** CIRCUIT_DPLL$(n.right, 0, h)$ **then** ▷ or 1 if $n.right$ is negated
14: **return** SAT
15: **end if**
16: $h \leftarrow h'$ ▷ undo assignments
17: **return** UNSAT

At the core of the algorithm is the implication procedure IMPLY (Algorithm 8) which is the circuit analog of the unit propagation procedure of clausal DPLL. In AIG-based and other systems with gate fanin set to a small constant, the implication procedure can be implemented very efficiently by means of lookup tables – Table 2.6 shows parts of such table schematically. In [Thiffault et al., 2004] and [Wu et al., 2007] the authors suggested using the watched literals scheme (Section 2.2.1) to allow efficient implication through gates with arbitrary fanin. Although the authors did claim that their approach results in faster propagation through gates with large fanin (as compared to the equivalent tree of binary AND gates), no experimental data to support this claim was presented. Note that for clarity we omitted the special treatment of the input nodes in Algorithm 8.

Algorithm 8 IMPLY([*in*] n, [*in*] ν, [*in, out*] h)

Input: n – node; ν – truth value for n; h – current truth-value assignment.
Output: FALSE – conflict; TRUE – no conflict; implied assignments are added to h.

1: $h \leftarrow h \cup \{n \mapsto \nu\}$
2: $\nu_{left} \leftarrow h(n.left)$ ▷ X if n is unassigned
3: $\nu_{right} \leftarrow h(n.right)$ ▷ X if n is unassigned
4: $s \leftarrow$ LOOKUP$(\nu, \nu_{left}, \nu_{right})$
5: **if** $s.action$ = CONFLICT **then**
6: **return** FALSE
7: **else if** $s.action$ = JUSTIFY **then**
8: $justification\_queue.enqueue(n)$
9: **return** TRUE
10: **else if** $s.action$ = PROP-LEFT-RIGHT **then**
11: **return** IMPLY$(n.left, s.lvalue, h)$ && IMPLY$(n.right, s.rvalue, h)$
12: **else if** ... **then**
13: ...
14: **end if**

The optimal strategy for selection of the next node from the justification queue (Algorithm 7, line 7) is largely dependent on a particular instance. Often depth-first selection produces good results, as it tends to generate conflicts quickly. Other strategies such as using a mixture of breath-first and depth-first, performing look-ahead (as in Section 2.2.2), or targeting nodes that are involved in small cuts of the circuit graph, were suggested as well [Kuehlmann, 2008]. Unfortunately, we were not able to find any more specific information or analysis of different selection strategies in circuit SAT

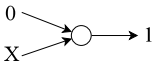
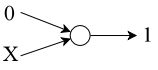
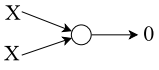
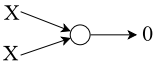
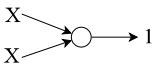
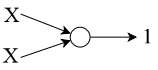
Current Assignment	Action	Next Assignment
	CONFLICT	
	JUSTIFY	
	PROP_LEFT_RIGHT	
...	...	...

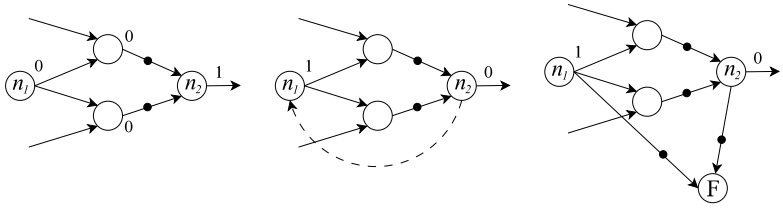
Table 2.1: Implication lookup table for AIG circuits

algorithms. In the ATPG literature, it is commonly accepted that a selection strategy based on *backtracing* [Fujiwara and Shimono, 1983, Hamzaoglu and Patel, 1998] produces the best results – curiously, we could not find any publications related to use of this strategy in circuit SAT solvers.

We now briefly describe some of the optimizations used in state-of-the-art circuit SAT solvers, all of which, perhaps with the exception of conflict-based learning, take advantage of the underlying circuit structure.

- *Static learning* is a preprocessing technique originally suggested in [Schulz et al., 1988] and designed to identify the so-called *indirect implications* in the circuit. An implication is direct if it can be derived using the implication procedure as in Algorithm 8. More specifically, let n_1 and n_2 be two nodes, ν_1 and ν_2 be truth-values, and h be an empty truth-value assignment. Then, if after the invocation $\textsc{Imply}(n_1, \nu_1, h)$, h contains assignment $n_2 \mapsto \nu_2$, we say that $(n_1 = \nu_1) \rightarrow (n_2 = \nu_2)$ is a direct implication. Consider now the circuit in Figure 2.7. Assignment $n_1 = 0$ directly implies assignment $n_2 = 1$ (see Figure 2.7(a)). Note, that $(n_1 = 0) \rightarrow (n_2 = 1)$ is equivalent to $(n_2 = 0) \rightarrow (n_1 = 1)$, but this latter implication could not be obtained by $\textsc{Imply}()$. In fact, an assignment of 0 to n_2 during search makes n_2 unjustified and causes branching (see Figure 2.7(b)). Once the indirect implication has been detected, it can be inserted into the circuit, as demonstrated in Figure 2.7(c) – note that now, assignment $n_2 = 0$ directly implies $n_1 = 1$.

  The algorithm for detection of indirect implications presented in [Schulz et al., 1988] picks a node from the circuit, assigns a truth-value to that node, performs direct implications and reverses them (as above), and uses a heuristic to determine which of the resulting implications are



(a) Direct implication

(b) Indirect implication

(c) Modified circuit

Figure 2.7: Static learning example. The node marked F in (c) is assigned to constant 0.

indirect. Thus, the algorithm is incomplete in a sense that it produces only a subset of the indirect implications. A complete algorithm for static learning called *recursive learning* is due to [Kunz and Pradhan, 1992].

- *Dynamic learning* also proposed in [Schulz et al., 1988] and subsequently improved in [Kunz and Pradhan, 1993] is, just like static learning, a process of deriving indirect implications. However, as opposite from static learning, dynamic learning is performed during the search, detecting implications that are valid only in the current part of the search space. Even though the learned implications cannot be added to the circuit structure permanently, they can still be used by the implication procedure.

- *Conflict-based learning*, in conjunction with non-chronological backtracking, as described in Section 2.2.1, was first incorporated into circuit SAT solvers in [Ganai et al., 2002]. The algorithm proposed by the authors maintains the conflict clauses in a separate clausal database – the motivation behind this decision is to combine the strengths of the fast implication algorithm of AIG-based solvers, and the fast unit propagation of the conflict-driven CNF solvers. Combining clause learning with a circuit-based decision heuristic resulted in a solver that consistently outperformed the best at a time CNF solver (`zChaff`), in some cases by an order of magnitude. More recently, the authors in [Wu et al., 2007] suggested inserting the conflict clauses into the circuit as OR gates. Combined with their improved watched literals scheme for arbitrary-fanin gates, their algorithm faired favorably compared

to some of state-of-the-art CNF solvers (e.g. `minisat`). However, no comparison with the hybrid technique of [Ganai et al., 2002] was presented.

- *Correlation detection* [Lu et al., 2003, 2004] is the idea of using *random simulation* of the circuit to detect relationships that are likely to hold between signals of the circuit. During random simulation a small set of random assignments to the inputs of the circuit is generated, and the values of internal nodes are recorded. If the recorded values show that, for example, node n_1 is assigned 0 whenever node n_2 is assigned 1, it is likely that $n_1 = \neg n_2$. Such "suspected" relationships are stored, and used during the search to guide decision heuristic – in our example, if n_1 is currently assigned 1, and the decision heuristic has a choice of the value to assign to n_2, it will choose 1 first with the intention to generate conflict as soon as possible. The authors in [Lu et al., 2003] also suggested to use the relationship information derived during simulation for the so called *incremental solving*. The idea is that if, for example, we suspect that $n_1 = \neg n_2$ for two nodes with small transitive fanin, we first try to solve $(n_1 = 1) \wedge (n_2 = 1)$ – since the problem is likely to be unsatisfiable, the process should generate many conflict clauses, which then can be used to solve the larger circuit. Thus, incremental solving can be seen as another form of learning (the authors call it *explicit learning*). In [Lu et al., 2004] both techniques are reported to result in significant performance gains.

2.7 Incomplete Algorithms for Circuits

To our knowledge, there is only one published attempt to develop an incomplete SAT solving algorithm for circuits. In [Muhammad and Stuckey, 2006] the authors presented an SLS-based algorithm SNCNFS which is a generalization of the original version of polSAT [Stachniak, 2002] to arbitrary boolean circuits. We omit the description of SNCNFS here due to the fact that polSAT is covered in detail in Section 2.5.1 – it worth mentioning, however, that the authors worked out the detailed calculations of clash values for various non-standard gates (for example XOR and "at-most-k" gate), and also given the explicit rules for calculation of "negative" clash values required for non-NNF formulas (in [Stachniak, 2002] only the rules for "positive" clash values were explicitly presented).

The results of the experimental study presented in [Muhammad and Stuckey, 2006] indicate that the algorithm performs well on circuits obtained

by encoding some of the crafted SAT problems. However, the circuits used in the study are of relatively small size and, perhaps most importantly, small depth. We emphasize this point, as our (unpublished) experiments on industrial-size circuits indicate that such a straightforward generalization of the polSAT algorithm does not work well.

2.8 Conclusion

To conclude this chapter we review some of the known "performance profiles" of the various classes of SAT algorithms. As it was already mentioned in Section 2.1, benchmark instances used to evaluate SAT algorithms are drawn from one of the following three categories – random, crafted and industrial. Perhaps the richest source of information on comparative performance of different classes of SAT algorithms are the results of SAT competitions. Unfortunately, SAT competitions are mostly designed specifically for CNF-based algorithms, and so there is no objective data for comparison of non-clausal and circuit-based algorithms. Nevertheless, we believe that the comparative strengths and weaknesses of different types of CNF-based algorithms could be indicative of the behavior of the algorithms based on the richer representation.

The performance profiles of different classes of CNF-based SAT algorithms are summarized in Table 2.8. For each class in the table, the "very good" entry indicates that this class generally outperforms all others, "good" indicates that this class performs well, "med" indicates that algorithms of this class are capable of solving medium difficulty problems, but fail on difficult ones, and, finally, "bad" indicates that performance of the algorithms from this class is unacceptable.

	Random		Crafted		Industrial	
	SAT	UNSAT	SAT	UNSAT	SAT	UNSAT
Conflict-driven DPLL	bad	bad	good	very good	very good	very good
Look-ahead DPLL	good	very good	very good	good	med	med
Stochastic local search	very good	n/a	med	n/a	bad	n/a
Unit propagation local search	good	n/a	good	n/a	med	n/a

Table 2.2: Performance profiles of SAT algorithms.

Please keep in mind that the table gives only a very high-level view – there are many cases when algorithms that generally perform poorly on a class of problems, solve particular problems from this class extremely well. For example, the SLS-based algorithm presented in [Pham and Gretton, 2007] was the first algorithm ever to handle a certain very difficult problem from the crafted category.

Nevertheless, some conclusions can be drawn even from such a generalized picture. In particular, it is clear that there is no one class of SAT algorithms that performs well across the broad spectrum of problem types. This observation motivates research in two, slightly overlapping, directions. One is the development of hybrid SAT algorithms, for example by integrating certain features of complete and incomplete algorithms. Some of the recent work in this direction is presented in [Jussien and Lhomme, 2000, Richards and Richards, 2000, Fang and Ruml, 2004, Habet and Vasquez, 2007, Fang and Hsiao, 2008]. Another promising research direction is the development of so-called *algorithm portfolios* which combine SAT solvers of different types and automatically determine which solver to run based on the properties of the SAT instance at hand. This line of research resulted in the portfolio-based SAT solver `SATZilla` which dominated in both the crafted and the random categories in the 2007 SAT competition. A recent paper [Xu et al., 2008] describes this approach in great detail.

Finally, with respect to non-clausal and circuit SAT algorithms, the results of various empirical studies published in [Ganai and Kuehlmann, 2000, Ganai et al., 2002, Lu et al., 2003, 2004, Thiffault et al., 2004, Jain et al., 2006, Muhammad and Stuckey, 2006, Wu et al., 2007, Stachniak and Belov, 2008] seem to indicate that SAT algorithms based on the enriched representation have the potential to outperform CNF-based algorithms, at least on some types of problems.

Chapter 3

Applications of SAT

3.1 Introduction

In this chapter we discuss some of the practical applications of SAT solving algorithms. One of the reasons for the incredible amount of attention devoted to the development of SAT algorithms is the applicability of SAT to a large variety of problems in hardware and software design and verification.

In this chapter we give detailed examples of two of such applications of SAT, namely in Bounded Model Checking (BMC) and in solving the Satisfiability Modulo Theories (SMT) problem. We selected these two applications for a number of reasons. First, both BMC and SMT are very general techniques and so are widely applicable both in hardware and in software verification applications. Second, the two techniques are based on very different principles: BMC works with models of computation, while SMT is purely logical. We thought it would be informative to demonstrate the way SAT is used in two such different applications. Third, SAT solving in both of these applications is geared towards finding satisfying assignments, rather than proving unsatisfiability. Thus, we believe both methods could benefit from incomplete SAT solving techniques, which are of interest to us.

The rest of this chapter is organized as follows: we start with the discussion of BMC in Section 3.2. We overview SMT in Section 3.3, and we finish this chapter in Section 3.4 with the listing of other practical applications of SAT and some concluding remarks.

3.2 Bounded Model Checking

Bounded model checking (BMC) is a SAT-based incomplete variant of a technique for verification of temporal properties of finite-state systems called *model checking* [Clarke and Emerson, 1982, Queille and Sifakis, 1982]. In model checking the system under consideration is *modeled* as a finite state transition system in which each state is associated (or, *labeled*) with some set of atomic propositions that hold in this state. Atomic propositions are arbitrary application-specific expressions that evaluate to true or false. For example, in the model of two concurrent processes a state could be labeled by a set $\{pc_0 = 2, has\_mutex_0, pc_1 = 3\}$.

Formally, such, labeled, finite state transition systems are represented by *Kripke structures*:

Definition 2 (Kripke structure). *Let $\mathscr{P}$ be a set of atomic propositions. A* Kripke structure *is a tuple $M = (S, I, T, L)$, where S is a finite set of states, $I \subseteq S$ is a set of initial states, $T \subseteq S \times S$ is a transition relation, and $L : S \mapsto 2^{\mathscr{P}}$ is a labeling function which associates with each state $s \in S$ a set $L(s)$ of atomic propositions that hold in s.*

To simplify the presentation, we will assume that for every $s \in S$, there exists s', such that $(s, s') \in T$, i.e. T is total. Additionally, we will assume that L is one-to-one, and so a state s is uniquely defined by its label $L(s)$ – note that in the general case, the states with identical labels can be collapsed into one.

Specifications, that is the properties to be verified, of systems in model checking are expressed by formulas of temporal propositional logic. Semantics of specification formulas depend on the view of time in the logic used. With linear time temporal logics, which view time as a sequence of points, the expectation is that every possible execution path of the system will adhere to the specification. With branching time temporal logics, in which each time point can have multiple successors, the expectation is that the (unique) computation tree of the system will adhere to the specification. There has been a considerable debate in the literature regarding the comparative strengths and weaknesses of linear and branching time logics. Even though a definitive conclusion has not been reached, in a most recent review on this subject Vardi [Vardi, 2001] gives a compelling argument for the advantages of the linear time view. In addition, the linear time view fits more naturally BMC than the branching time view. Thus, in this section, we will focus on a logic called *Linear Temporal Logic (LTL)*, which, as the name suggests, adopts the linear time view.

The syntax of LTL is defined in the following way:

Definition 3 (LTL syntax). *Let $\mathscr{P}$ be a set of atomic propositions. The set of LTL formulas is a smallest set $\mathscr{L}$ such that*

- $\mathscr{P} \subseteq \mathscr{L}$;
- *If $\alpha \in \mathscr{L}$, and $\beta \in \mathscr{L}$, then so are $(\neg\alpha)$, $(\alpha \wedge \beta)$, $(\mathbf{X}\alpha)$ – pronounced "next α", $(\alpha \mathbf{U} \beta)$ – pronounced "α until β".*

The semantics of LTL formulas are given in terms of sequences of states, or *paths*, in a Kripke structure: given a Kripke structure $M = (S, I, T, L)$, a *path* is a infinite sequence $\pi = (s_0, s_1, \dots)$ that respects the transition relation T, i.e. for all i, $s_{i+1} \in T(s_i, s_{i+1})$[1]. If $s_0 \in I$, then path is called *initialized*. Given a path $\pi = (s_0, s_1, \dots)$, we will denote s_i as π_i, the subsequence $(s_i, \dots, s_j)$, for $i \leq j$ as $\pi_{i,j}$, and the suffix $(s_i, \dots)$ of π as $\pi_{i,\infty}$. Note that if π is a path, then so is $\pi_{i,\infty}$ for all $i \geq 0$.

Definition 4 (LTL semantics). *Let $\mathscr{P}$ be a set of atomic propositions, $M = (S, I, T, L)$ be a Kripke structure, and π be a path in M. An LTL formula α* holds on *π, in symbols, $\pi \models \alpha$, if the following conditions hold:*

- *if $\alpha = p$ for some $p \in \mathscr{P}$, then $p \in L(\pi_0)$;*
- *if $\alpha = (\neg\beta)$, then β does not hold on π ($\pi \not\models \beta$);*
- *if $\alpha = (\beta \wedge \gamma)$, then $\pi \models \beta$ and $\pi \models \gamma$;*
- *if $\alpha = (\mathbf{X}\beta)$, then $\pi_{1,\infty} \models \beta$;*
- *if $\alpha = (\beta \mathbf{U} \gamma)$, then there exists $i \geq 0$, such that for all j, $0 \leq j < i$, $\pi_{j,\infty} \models \beta$, and $\pi_{i,\infty} \models \gamma$.*

An LTL formula α holds in *M, in symbols, $M \models \alpha$, iff α holds on all initialized paths of M. Finally, two LTL formulas α and β are* equivalent, *in symbols, $\alpha \equiv \beta$, iff for every Kripke structure M, $M \models \alpha$ iff $M \models \beta$.*

Thus, the intuitive meaning of $\mathbf{X}\alpha$ is that α has to be true in the next state, and the intuitive meaning of $\alpha \mathbf{U} \beta$ is that α has to be true until β becomes true (and β must become true at some point).

[1] Earlier we made an assumption that the transition relation T of Kripke structures is total, and so the systems modeled by such structures do not terminate. Since paths are intended to model execution sequences, it makes sense to require for paths to be infinite. Some authors add a set of terminating states F to the Kripke structure, and require all paths to be maximal (i.e. either ending in terminating state or infinite).

Additional propositional connectives ($\vee$, $\rightarrow$, $\leftrightarrow$, etc.) as well as logical constants T and F are introduced into LTL by usual means. Additional LTL-specific operators can defined in the following way.

- $\mathbf{F}\alpha \triangleq (T\mathbf{U}\alpha)$

  Pronounced "finally α". Thus, $\pi \models (\mathbf{F}\alpha)$ if for some $i \geq 0$, $\pi_{i,\infty} \models \alpha$. The intuitive meaning of $\mathbf{F}\alpha$ is that α has to be true at some state now or in the future.

- $\mathbf{G}\alpha \triangleq \neg(\mathbf{F}(\neg\alpha))$

  Pronounced "globally α". Thus, $\pi \models (\mathbf{F}\alpha)$ if for all $i \geq 0$, $\pi_{i,\infty} \models \alpha$. The intuitive meaning of $\mathbf{G}\alpha$ is that α has to be true in all states.

Its worth to note that alternative notation and naming for LTL-specific operators is common: $\circ\alpha$ instead of $\mathbf{X}\alpha$, $\Diamond\alpha$ (pronounced "eventually") instead of $\mathbf{F}\alpha$, and $\Box\alpha$ (pronounced "always") instead of $\mathbf{G}\alpha$.

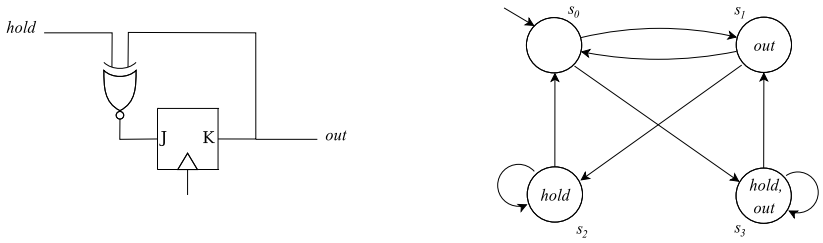
When $M \models \alpha$ we often say that α is *universally valid* in M, because α is required to hold on all initialized paths in M. Sometimes it is convenient to require that α holds only on some initialized path in M. We say that α is *existentially valid* in M if there exists an initialized path π in M such that $\pi \models \alpha$. Although in the literature, authors often borrow operators $\mathbf{A}$ and $\mathbf{E}$ from Computational Tree Logic (CTL) to denote universal validity by $M \models \mathbf{A}\alpha$, and existential validity by $M \models \mathbf{E}\alpha$, we feel that $M \models_\forall \alpha$ and $M \models_\exists \alpha$ is more appropriate choice. Clearly, $M \models_\forall \alpha$ if and only if $M \not\models_\exists \neg\alpha$.

LTL model checking problem can now be stated formally as: given a Kripke structure M and an LTL formula α, determine whether $M \models_\forall \alpha$.

As an example we consider a problem of verification of a synchronous circuit design depicted in Figure 3.1(a)[2]. Assume that the flip-flop reads the new input on the clock rise, and outputs the new value on the clock fall. As it is common in model checking of synchronous circuits, we will ignore the clock and the timing issues, and so we will assume that the input *hold* is provided well enough in advance. The circuit implements a one-bit counter with hold – when *hold* line is reset, the output alternates between 0 and 1, otherwise when *hold* is set the output is fixed at the most recent output value.

The state of the system is completely described by the values of *hold* and *out* on clock rise. Since both values are binary, we let $\mathscr{P} = \{hold, out\}$. Figure 3.1(b) depicts the Kripke structure M of the system. Note that since

[2] A reader unfamiliar with the hardware circuit notation may safely skip this paragraph.



(a) Circuit diagram (b) Kripke structure

Figure 3.1: The one bit counter with hold.

hold is an external input, every state has two successors, reflecting the fact that the input value may or may not change between the clock rises.

The following illustrates the use of LTL for specification of properties of this system. Note that here, and in the remainder of this section, we will omit brackets in LTL formulas when no ambiguity may arise.

- Formula $\alpha_1 = \mathbf{G}(\neg hold \rightarrow (out \leftrightarrow \mathbf{X}\neg out))$ specifies that as long as *hold* is reset, *out* changes from 0 to 1 on each subsequent clock. This is an example of a so called *safety* property – intuitively, a property that "something bad does not happen". It is not difficult to see that $M \models_\forall \alpha_1$.

- Formula $\alpha_2 = \mathbf{G}((hold \leftrightarrow \mathbf{X}\neg hold) \rightarrow \mathbf{F}(out \leftrightarrow \mathbf{X}\neg out))$ asserts that if *hold* changes at certain point, then *out* must also eventually change. This is an example of a *liveness* property – intuitively, a property that "something good eventually happens". Note that $M \models_\forall \alpha_2$.

- Formula $\alpha_3 = \mathbf{F}(hold \wedge out)$ asserts that eventually *hold* and *out* will be both set. This is also an example of a liveness property. In this case $M \not\models_\forall \alpha_3$, but $M \models_\exists \alpha_3$.

Safety and liveness properties are among the most common properties used with model checking. In fact, every LTL formula is equivalent to a conjunction of a formula that expresses safety property and a formula that expresses a liveness property [Alpern and Schneider, 1985].

A classic algorithm for LTL-based model checking [Holzmann, 1991] is based on the observation that both the Kripke structure and LTL formulas can be associated with a computational model called *Buchi automaton*. A

Buchi automaton is a non-deterministic finite automaton extended to accept infinite words. Languages recognized by Buchi automata are called ω-regular languages. Each path π in a given Kripke structure $M = (S, I, T, L)$ can be described by an infinite word over an alphabet $\Sigma = 2^{\mathscr{P}}$, where $\mathscr{P}$ is the set of atomic propositions. The i-th symbol in this word is the set $L(\pi_i)$. Thus, the set of all initialized paths of M constitutes a language $L(M)$ over Σ. In a similar manner, given an LTL formula α, the set of all paths in which α holds can be associated with the language $L(\alpha)$. Hence, in order to determine whether $M \models_\forall \alpha$ one can check whether $L(M) \subseteq L(\alpha)$, or, equivalently, whether $L(M) \cap L(\neg\alpha) = \emptyset$. Both $L(M)$ and $L(\alpha)$ can be shown to be ω-regular languages (see [Clarke and Schlingloff, 2001]), and therefore the latter condition can be verified by testing the emptiness of the product automaton of the Buchi automaton for $L(M)$ and the Buchi automaton for $L(\neg\alpha)$.

The automata for both M and $\neg\alpha$ can be constructed on-the-fly [Courcoubetis et al., 1992, Hammer et al., 2005] thereby avoiding building the whole product automaton which can contain unreachable states. Nevertheless, in many industrial applications, particularly in model checking of hardware designs, the size of the reachable state space is prohibitive – consider, for example, the exponential in n size of the reachable state space of an n-bit counter.

Symbolic model checking introduced in early 1990's by Burch et al. [Burch et al., 1990] addresses this so called *state explosion problem* by recording and manipulating sets of states instead of individual states. The basic principle is as follows. As usual, let $\mathscr{P}$ be a set of atomic propositions, and let $M = (S, I, T, L)$ be a Kripke structure. Consider the set $\mathscr{P}' = \bigcup_{s \in S} L(s)$. Assuming that $|\mathscr{P}'| = n$, one can represent each state $s \in S$ by a Boolean vector $\vec{v} \in \{0, 1\}^n$ – each element of this vector corresponds to a particular $p \in \mathscr{P}'$, and is set to 1 if $p \in L(s)$, and 0 otherwise. This way, each state s can be described by a truth-value assignment to the variables in $\vec{v}$. Given such a truth-value assignment, one can write a Boolean formula $\psi_s(\vec{v})$ which is true for exactly that assignment, and so can be used to *encode* s. Furthermore, since a Boolean formula can have many satisfying assignments, given a set of states $S' \subseteq S$, one can construct a formula $\psi_{S'}(\vec{v})$ which encodes the states in S'. By adding a second vector of propositional variables, $\vec{w}$, with the intended purpose of representing the "next state", one can also write a Boolean formula $\psi_T(\vec{v}, \vec{w})$ which encodes the transition relation T – every truth-value assignment to variables from $\vec{v}$ and $\vec{w}$ which satisfies $\psi_T(\vec{v}, \vec{w})$

encodes two states s and s' such that $(s, s') \in T$. In this manner, the formula

$$\psi_I(\vec{w}) \vee \exists \vec{v}\ (\psi_I(\vec{v}) \wedge \psi_T(\vec{v}, \vec{w}))$$

for example, represents the set of states reachable from I in at most one step[3].

Symbolic model checkers use Reduced Ordered Binary Decision Diagrams (ROBDDs, or often, though incorrectly, simply BDDs) [Bryant, 1986, 1992] to represent and manipulate Boolean formulas efficiently. A BDD for a propositional formula can be seen as a compact representation of binary decision tree of the Boolean function represented by that formula. Given a fixed ordering on variables of the formula, it can be shown that this representation is unique. Hence, BDDs are a canonical form for Boolean formulas. BDDs are often substantially more compact than traditional normal form representations such as CNF or DNF. The basic logical operations of negation, conjunction, disjunction, substitution for variable, and projection ($\exists p \alpha(p)$) on Boolean formulas can still be implemented efficiently when formulas are represented with BDDs.

Thus, if, for example, α is an LTL formula without temporal connectives, then in order to verify $M \models \mathbf{G}\alpha$ (an *invariant* property) a naive symbolic model checking algorithm would construct a BDD of the formula $\psi(\vec{v})$ which represents the set of all reachable states, a BDD of the formula $\psi_\alpha(\vec{v})$ which represents the set of states in which α holds, and check if the BDD of the formula $\psi(\vec{v}) \rightarrow \psi_\alpha(\vec{v})$ represents a tautology.

Since its introduction, symbolic model checking techniques based on BDDs have been extremely successful and gained wide adoption in industry. Unfortunately, as the complexity of designs grew, BDD based methods have begun to find their limitations. The main issue is the amount of memory required to store BDDs – this issue is particularly prominent in verification of synchronous hardware and concurrent software systems. In addition, the BDD size is extremely sensitive to the ordering of the variables, and so BDD based symbolic model checkers often require a large amount of application-specific manual tuning.

Like symbolic model checking, *bounded model checking (BMC)*, introduced in [Biere et al., 1999], is based on the idea of casting the LTL model checking problem into propositional logic. In BMC, given a Kripke structure M, an LTL formula α and a *bound* $k \in \mathbb{N}$ we construct a propositional formula $[[M, \alpha]]_k$ with the following property. If $[[M, \alpha]]_k$ is satisfiable, then every

[3]The expression $\exists p \alpha(p)$ is an abbreviation for $\alpha(p/F) \vee \alpha(p/T)$

satisfying truth-value assignment represents a finite prefix $\pi_{0,k}$ of a path π in M such that $\pi_{0,k}$ alone is enough to guarantee that $\pi \models \alpha$. Thus, a satisfying assignment represents a *witness* of length k to the fact that α is existentially valid in M. Conversely, if $[[M, \alpha]]_k$ is not satisfiable then no witness of length k exists.

Hence, given an LTL formula α, and a Kripke structure M, a BMC-based model checker uses a SAT solver to test the satisfiability of formulas $[[M, \neg\alpha]]_k$ for sequentially increasing values of k until one of the following occurs:

1. $[[M, \neg\alpha]]_k$ is satisfiable for some k. This implies that $M \models_\exists \neg\alpha$, and therefore $M \not\models_\forall \alpha$. Since a satisfying assignment is a witness for the existential validity of $\neg\alpha$ it provides a *counterexample* to the universal validity of α in M – in other words, a path in M which violates α.

2. $[[M, \neg\alpha]]_k$ is unsatisfiable. If k reached the so called *completeness threshold*, (see below) then the model checker declares that $M \models_\forall \alpha$, otherwise k is incremented and $[[M, \neg\alpha]]_k$ is checked for satisfiability again.

   Given a Kripke structure M and an LTL formula α, the *completeness threshold* for M and α is a natural number ct such that if no witness of length ct or less exists, then $M \not\models_\exists \alpha$. In [Biere et al., 1999] authors show that ct is at most $|S| \cdot 2^{|\alpha|}$, however for realistic applications this value is not of practical use. Computing the smallest value of ct is at least as hard as deciding whether $M \models \alpha$ [Clarke et al., 2004], and so several authors have developed techniques for computing useful over-approximations of ct for some simple classes of LTL formulas [Biere et al., 1999, Kroening and Strichman, 2003, Clarke et al., 2004].

3. The formula $[[M, \neg\alpha]]_k$ becomes too difficult to handle by the SAT solver. In this case, the model checker terminates without giving a definitive answer – all we know is that no execution of length k or less violates the property specified by α.

In many realistic applications the completeness threshold can rarely be reached, and so BMC is used for finding bugs rather than for the model checking proper.

Given a Kripke structure $M = (S, I, T, L)$, an LTL formula α and a bound k, the formula $[[M, \alpha]]_k$ may be constructed in several different ways. We give

an example of such a construction inspired by the method proposed in [Latvala et al., 2004]. The details of the original method, as well as proofs of the theoretical underpinnings of BMC are available in [Biere et al., 1999]. Also of interest is an automata-based construction developed in [Clarke et al., 2004].

Consider the Kripke structure M of the one bit counter with hold presented earlier in this section (see Figure 3.1). Although $\mathbf{F}(\mathit{hold} \wedge \mathit{out})$ is not a desirable property of the system, for the sake of example, let us assume that it is, and so that we would like to check whether $M \models_\forall \mathbf{F}(\mathit{hold} \wedge \mathit{out})$, or equivalently whether $M \models_\exists \mathbf{G}\neg(\mathit{hold} \wedge \mathit{out})$. A witness for existential validity of $\mathbf{G}\neg(\mathit{hold} \wedge \mathit{out})$ would constitute a counterexample for universal validity of $\mathbf{F}(\mathit{hold} \wedge \mathit{out})$.

Let $\vec{s} = \{h, o\}$ be the propositional representation of the states in M, in which h corresponds to atomic the proposition *hold*, and o corresponds to the atomic proposition *out*. Then, the formula

$$I(h, o) = \neg h \wedge \neg o$$

represents the set of initial states in M, and the formula

$$\begin{aligned} T(h, o, h', o') = {} & (\neg h \wedge \neg o) \rightarrow o' \wedge \\ & (\neg h \wedge o) \rightarrow \neg o' \wedge \\ & (h \wedge \neg o) \rightarrow \neg o' \wedge \\ & (h \wedge o) \rightarrow o' \end{aligned}$$

represents the transition relation of M.

In BMC we are looking for a witness of bounded length k. Let $k = 2$ be the current bound (i.e. no witness of length $k = 0$ and $k = 1$ has been found), and let $\vec{s_0}, \vec{s_1}, \vec{s_2}$ be the three states of the witness, starting from initial state $\vec{s_0}$. If each state $\vec{s_i}, i = 0, 1, 2$ is encoded by two variables h_i, o_i, the formula

$$I(h_0, o_0) \wedge T(h_0, o_0, h_1, o_1) \wedge T(h_1, o_1, h_2, o_2) \tag{3.1}$$

expresses the constraint that these three states do indeed constitute a prefix of a valid path in M.

For any LTL formula α, any finite prefix that is a witness for $\mathbf{G}\alpha$ must contain a loop. Thus, we require that state $\vec{s_2}$ has to be the same as either state $\vec{s_0}$ or $\vec{s_1}$. The formula

$$(h_2 \leftrightarrow h_0 \wedge o_2 \leftrightarrow o_0) \vee (h_2 \leftrightarrow h_1 \wedge o_2 \leftrightarrow o_1) \tag{3.2}$$

expresses this *loop constraint*.

Finally, we have to make sure that the states of the witness fulfill the requirements imposed by the formula $\mathbf{G}\neg(hold \wedge out)$ – that is, in every state, $\neg(hold \wedge out)$ has to hold:

$$\neg(h_0 \wedge o_0) \wedge \neg(h_1 \wedge o_1) \wedge \neg(h_2 \wedge o_2). \tag{3.3}$$

The conjunction of formulas (3.1), (3.2), and (3.3) is the formula $[[M, \mathbf{G}\neg(hold \wedge out)]]_2$, which now is checked for satisfiability. The formula is satisfiable: the assignment $\{h_0 = 0, o_0 = 0, h_1 = 0, o_1 = 1, h_2 = 0, o_2 = 0\}$ is satisfying. This assignment represents a finite prefix (s_0, s_1, s_0) of a path $(s_0, s_1, s_0, s_1, \dots)$ that violates the property $\mathbf{G}(hold \wedge out)$.

To conclude this section, we would like to make a few observations. First, we would like to point out that SAT formulas created during BMC contain many identical subformulas (this can be seen even in our simple example) and therefore most naturally represented as DAGs. Furthermore, the formulas contain many structurally identical subformulas – consider for example the multiple instantiations of formula $T(h, o, h', o')$. Thus, we believe that in the context of BMC, SAT solvers that work directly on DAGs could be of advantage compared to CNF-based solvers. Although to our knowledge no comparative studies to support or refute this claim have been performed, the results of experimental evaluation of circuit SAT solvers on BMC benchmarks presented in [Ganai et al., 2002] and [Wu et al., 2007] seem to be promising.

The second point we would like to make is that due to the fact BMC is used almost exclusively as a bug finding technique, in the context of BMC we are more interested in finding satisfying assignments, rather than proving that the formula is not satisfiable. Thus, we believe that there is a place for incomplete SAT solvers in BMC – one can envision a system where an incomplete solver works in parallel with a complete solver, with the former looking for satisfying assignments, and the latter working on proving unsatisfiability.

3.3 Satisfiability Modulo Theories (SMT)

Satisfiability Modulo Theories (SMT) is a problem of determining the satisfiability of a first-order logic (FOL) formula in one or more *background theories*. Typically, the formulas are quantifier-free and the theories of interest are those of integers, reals, arrays, recursive datatypes, bitvectors,

and uninterpreted functions. As an example, consider the following formula

$$P \wedge (x \leq y) \wedge ((select(a, x) = f(y)) \rightarrow \neg Q \vee (select(a, y) = f(x))), \quad (3.4)$$

where x, y and $\leq$ are from the theory of integers, P, Q and f are uninterpreted symbols[4], and *select* and a are from the theory of arrays.

Formally, given a set of FOL theories $T_1, \ldots, T_n$ over the languages $\mathscr{L}_1, \ldots, \mathscr{L}_n$, respectively, and a quantifier-free FOL formula α over the language $\mathscr{L} = \mathscr{L}_1 \cup \cdots \cup \mathscr{L}_n$, the task is to determine whether there exist a model M of the theory $T = T_1 \cup \cdots \cup T_n$ such that $M \models T \cup \alpha[s]$ for some assignment s to the free variables of α. If the answer is affirmative (α is called *T-satisfiable* in this case), we are typically interested to obtain the variable assignment s, as well as the interpretation of uninterpreted symbols of $\mathscr{L}$ that appear in α. The languages $\mathscr{L}_i$ are assumed to include $=$, and, with the exception of $=$, assumed to be disjoint. All of the theories are assumed to include the axioms asserting that $=$ is a congruence relation.

We now list some of the theories frequently used in various hardware and software verification applications. The approach to solving SMT described in this section relies on the availability of decision procedures for theories of interest, and so we provide references to the publications which contain descriptions of such procedures. Additional information is available from [Manna and Zarba, 2003] and [Sebastiani, 2006].

The theory of *Equality and Uninterpreted Functions (EUF)* is a first-order theory over the language $\mathscr{L}_{EUF} = \langle f_1, \ldots, f_k; P_1, \ldots P_m \rangle$, with $k, m \geq 0$, and symbols f_i, P_j of arbitrary arity. As the name suggests, the predicate and function symbols are left uninterpreted, and so the theory is axiomatized by specifying only that $=$ is a congruence relation. Although EUF is undecidable (due to undecidability of first-order logic), the quantifier-free fragment of EUF is decidable in polynomial time [Ackermann, 1954]. A modern efficient algorithm can be found in [Nieuwenhuis and Oliveras, 2005]. EUF formulas arise in hardware verification (for example, in processor control verification in [Burch and Dill, 1994]).

The theory of *Linear Integer Arithmetic (LA($\mathbb{Z}$))*, also known as *Presburger Arithmetic* is the set of all sentences over the language $\mathscr{L}_{LA} = \langle +, 0, 1; \leq \rangle$ that are valid in the structure with the set $\mathbb{Z}$ as domain and the standard interpretation of the rest of the symbols. The integer numerals, the

[4]Propositional variables are uninterpreted 0-ary predicate symbols, and so in this section they will be denoted by capital letters.

rest of the relations ($<, >, \geq, \neq$), and multiplication by an integer constant can be introduced via abbreviations. $LA(\mathbb{Z})$ is decidable – this is the famous result of [Presburger, 1930]. The decision procedure for $LA(\mathbb{Z})$ has triple-exponential time complexity [Oppen, 1973], and cannot be improved, unless $\mathcal{P} = \mathcal{NP}$ [Fischer and Rabin, 1974]. The decision problem for the quantifier-free fragment of LA($\mathbb{Z}$) – commonly known as the integer programming problem – is $\mathcal{NP}$-complete [Papadimitriou, 1981]. Nevertheless, algorithms that work well in practice are available (for example, Omega Test [Pugh, 1991]). Some applications that produce $LA(\mathbb{Z})$ formulas are verification of timing diagrams [Amon et al., 1997], buffer overrun detection in C code [Wagner et al., 2000], and RTL datapath verification [Brinkmann and Drechsler, 2002].

The theory of *Linear Real Arithmetic (LA($\mathbb{R}$))* is, similarly to LA($\mathbb{Z}$), the set of all valid $\mathscr{L}_{LA}$ sentences, but this time over the structure with $\mathbb{R}$ as domain. As with LA($\mathbb{Z}$), the integer numerals, the relations $<, >, \geq, \neq$, and multiplication by an integer constant can be introduced via abbreviations. Rational constants, and multiplication by rational constant are allowed in formulas as well, as every such formula can be rewritten into an equivalent formula that involves only integer constants and multiplication by integer. LA($\mathbb{Z}$) is decidable – in fact, the decidability result holds for the larger theory which has the operation of multiplication in the language [Tarski, 1948]. Decision procedure for LA($\mathbb{R}$) [Ferrante and Rackoff, 1975] has double-exponential runtime, and cannot be improved, unless $\mathcal{P} = \mathcal{NP}$ [Fischer and Rabin, 1974]. The quantifier-free fragment of LA($\mathbb{R}$), known as linear programming, is decidable in polynomial time [Khachiyan, 1979], although worst-case exponential algorithms, such as Simplex [Nelson, 1981] seem to work better in practice. LA($\mathbb{R}$) formulas arise in software verification [Dellacherie et al., 1999] and test pattern generation for hardware designs [Fallah et al., 2001].

The *Unit-Two-Variable-Per-Inequality (UTVPI)* theory is a syntactic fragment of LA($\mathbb{Z}$), in which the allowed form of atomic formulas is $\pm x \pm y \leq c$, where x, y are variables, and $c \in \mathbb{Z}$ is a constant. Being subset of LA($\mathbb{Z}$), UTVPI is decidable, and, as opposed to full linear integer arithmetic, the quantifier-free fragment of UTVPI is decidable in polynomial time [Harvey and Stukey, 1997], and very efficient practical algorithms are available [Lahiri and Musuvathi, 2005]. UTPVI formulas often comprise a large portion of the sets of linear integer arithmetic formulas that arise in verification applications (see [Ball et al., 2004] for an example from symbolic model

checking).

The theory of *Integer Difference Logic (DL(ℤ)* is a further restricted fragment of UTVPI. Atomic formulas of DL(ℤ) are of the form $x - y \leq c$. Efficient algorithms specialized to decide the satisfiability quantifier-free DL(ℤ) formulas are available [Cotton and Maler, 2006]. As with UTPVI, DL(ℤ) formulas comprise a large portion of linear inequalities produced in program verification [Pratt, 1977, Detlefs et al., 2005].

Other theories with decidable quantifier-free fragments that are of interest in software and hardware verification are the theory of *Bit Vectors (BV)* [Bozzano et al., 2006, Bruttomesso et al., 2007], the theory of *Arrays (AR)* [Stump et al., 2001], and the theory of *Recursive Datatypes (RDT)* [Bonacina and Echenim, 2007].

Algorithms for solving SMT can be obtained via a combination of theory-specific decision procedures with algorithms for SAT. This class of SMT algorithms was proposed in [Barrett et al., 2002] and [Flanagan et al., 2003], at the time when the solving power of SAT algorithms has dramatically increased due to the introduction of various optimization techniques discussed in Section 2.2.

To check the *T-satisfiability* of a quantifier-free FOL $\mathscr{L}$-formula α, the basic idea is to construct a *Boolean abstraction* α_B of α by replacing each atomic formula in α with a new propositional variable. If α_B is (propositionally) unsatisfiable then α is T-unsatisfiable. Otherwise, a (partial) satisfying assignment τ_B for α_B can be mapped into a conjunction of atomic $\mathscr{L}$-formulas τ, T-satisfiability of which can be checked by the decision procedure for T (so called *T-solver*). If τ is T-satisfiable, we are done. Otherwise, a clause prohibiting τ_B is added to α_B, and the satisfiability of the new formula is checked again.

To continue, we introduce the following notation[5]. Let $Props(\alpha)$ be a set of propositional variables in α, $Atoms(\alpha)$ the set of atomic formulas in α other than those in $Props(\alpha)$, and V a set of auxiliary propositional variables $\{V_1, \ldots, V_n\}$ such that $|V| = |Atoms(\alpha)|$ and $V \cap Props(\alpha) = \emptyset$. To construct the Boolean abstraction of α we define a bijection $t2b : Atoms(\alpha) \mapsto V$ ("theory-to-Boolean"). We denote the inverse bijection $t2b^{-1}$ by $b2t$ ("Boolean-to-theory"). Then, α_B is obtained from α by the simultaneous replacement of all atomic formulas ϕ in α with $t2b(\phi)$. Similarly, given

[5]Inspired by [Sebastiani, 2006].

a, possibly partial, truth-value assignment τ_B to variables in V, the corresponding conjunction τ of atomic $\mathscr{L}$-formulas is defined as

$$\bigwedge_{V_i \in V \text{ and } \tau_B(V_i) \text{ defined}} \quad \text{if } \tau_B(V_i) = 1 \text{ then } b2t(V_i) \text{ else } \neg b2t(V_i).$$

As an example, consider the formula α from (3.4):

$$P \wedge (x \leq y) \wedge ((select(a, x) = f(y)) \rightarrow \neg Q \vee (select(a, y) = f(x))).$$

Then, the map $t2b$ is defined as

$$\begin{aligned} x \leq y &\mapsto V_1 \\ select(a, x) = f(y) &\mapsto V_2 \\ select(a, y) = f(x) &\mapsto V_3, \end{aligned}$$

and $\alpha_B = P \wedge V_1 \wedge (V_2 \rightarrow \neg Q \vee V_3)$. A partial truth-value assignment that satisfies α_B is $\tau_B = \{P \mapsto 1, V_1 \mapsto 1, V_2 \mapsto 0\}$, and the corresponding conjunction τ of atomic formulas is $(x \leq y) \wedge \neg(select(a, x) = f(y))$.

The SMT algorithm outlined in the previous paragraph is presented in Algorithm 9. To implement the T-solver the algorithm relies on the procedure T_Solve([*in*] τ, [*out*] S), which, given a conjunction τ of atomic $\mathscr{L}$-formulas, returns UNSAT if h is not T-satisfiable, and SAT otherwise, in which case S contains the required variable assignment and the interpretation of uninterpreted symbols in τ. Additionally, the SMT algorithm presented in Algorithm 9 uses a complete SAT solver implemented by the procedure SAT_Solve([*in*] α_B, [*out*] τ_B).

To make the SMT-Solve algorithm efficient in practice, a T-solver should posses the following properties:

- *Conflict Set Generation* – whenever the T-solver determines that a conjunction τ is T-unsatisfiable, it is capable of returning a subset $\tau' \subset \tau$ which caused T-unsatisfiability – this is the *conflict set*. Then, the clause *cl* constructed in line 7 of Algorithm 9 can be built from τ'_B instead of τ_B. In practice, conflict sets are often significantly smaller than the initial conjunction, resulting in shorter clauses and a speed-up of the SAT solving step.

- *Incrementability* – often there is a significant overlap between the conjunctions τ passed in consecutive calls to the T-solver. Hence, it is desirable for a T-solver to be able to keep state between invocations.

Algorithm 9 SMT_Solve([*in*] α, [*out*] S)

Input: α – a quantifier-free $\mathscr{L}$-formula
Output: SAT and S if α is T-satisfiable; UNSAT – otherwise.

construct α_B from α
while SAT_Solve(α_B, τ_B) = SAT **do**
    construct τ from τ_B
    if T_Solve(τ, S) = SAT **then**
        return SAT
    end if
    $cl = \bigvee_{V_i \in V \text{ and } \tau_B(V_i) \text{ defined}}$ if $\tau_B(V_i) = 1$ then $\neg V_i$ else V_i
    $\alpha_B \leftarrow \alpha_B \wedge cl$
end while
return UNSAT

Incrementability is also desirable in the SAT solver, as only one extra clause is added on every invocation of the solver.

The approach to integration of SAT solving algorithms and T-solvers outlined in Algorithm 9 is called *offline integration*, as in this approach the SAT solver is treated as a black-box. In the *online integration* approach, the search for an assignment that satisfies the Boolean abstraction is integrated with the search for the T-assignment in one monolithic framework. Such tight integration allows to perform various optimizations. For example, one can check T-satisfiability of τ *during* the search for τ_B – in many cases calls to the T-solver allow to terminate search paths that will produce unsatisfiable conjunctions τ before the SAT solver completes the construction of τ_B. This optimization technique is called *early pruning*. Another idea that works well in practice is to integrate the unit propagation procedure in the SAT solver with reasoning in the theory T – this is so called *theory propagation*. For example, even though an α_B may not contain a clause $\neg V_1 \vee \neg V_2 \vee V_3$, where $p2t(V_1) = (x \leq y)$, $p2t(V_2) = (x \geq y)$, and $p2t(V_3) = (x = y)$, a theory propagation procedure will be able to deduce the assignment $\{V_3 \mapsto 1\}$ given an assignment $\{V_1 \mapsto 1, V_2 \mapsto 1\}$, and pass it to the SAT solver. Note that incrementability of the T-solver is essential for these two optimization techniques. There are many other optimizations of this kind – we refer to [Sebastiani, 2006] for detailed exposition of the online integration approach.

To conclude this section, we would like to draw the reader's attention to the fact that even though the completeness of SAT algorithms in SMT applications is required for proving unsatisfiability, the search is mostly geared

towards the *satisfying* assignments. This suggests that it may be possible to integrate efficient incomplete SAT algorithm into the SMT framework. The fact that SLS-based incomplete algorithms operate on complete (rather than partial) truth-value assignments may make this class of incomplete algorithms not appropriate in the SMT setting, as large conjunctions of atomic formulas would need to be handed off to T-solvers (though, incrementability of the T-solver may be able to compensate for that). Hence, the development of incomplete SAT algorithms that operate on *partial* truth-value assignments could be of interest – one such algorithm was proposed in [Prestwich, 2000, 2002].

3.4 Other Applications of SAT

Over the past forty years, SAT solvers have been used in a myriad of applications. Below we list some of the applications that are relevant to the current state-of-the-art in Computer Science and its industry. For each of the listed applications we give a short description (when appropriate), and provide a reference to the early publications which contain the initial ideas, as well as to some of the publications that describe the more recent developments.

- *Automatic Test Pattern Generation (ATPG)* is a technique used in hardware verification. Given a description of a circuit and a location of a possible fault, the goal of ATPG is to find a pair of assignments to the inputs of the circuit that will allow to distinguish correctly manufactured circuits from the faulty ones. Applications of SAT in ATPG are described in [Larrabee, 1992], [Marques-Silva and Sakallah, 1997], [Biere and Kunz, 2002].

- *Equivalence Checking (EC)* is also a hardware verification technique. The goal of EC is to ensure that two different circuit designs have the same functionality. The problem arises during the hardware design process as the high-level specification (*Register-Transfer Level*, or *RTL*) gets transformed in multiple stages into a low-level design. EC is used to ensure that the functionality is unaltered between the stages. Applications of SAT in EC are described in [Marques-Silva and Glass, 1999], [Goldberg et al., 2001], [Disch and Scholl, 2007]

- *Logic Synthesis* is a term that refers to a transformation from a higher-level design to a lower-level design in the hardware design process. Some of the applications of SAT to logic synthesis are described in

[Gu and Puri, 1995], [Wood and Rutenbar, 1998], [Khomenko et al., 2006], [Safarpour et al., 2006].

- *Scheduling* is a classic combinatorial optimization problem. Applications of SAT to scheduling are described for example in [Crawford and Baker, 1994], [Memik and Fallah, 2002], [Zhang et al., 2004].

- *AI Planning* is another classic problem. SAT can be used in planning to derive bounded-length plans – an approach somewhat similar to BMC. Applications of SAT to planning are described in [Kautz and Selman, 1992], [Rintanen et al., 2006], [Hoffmann et al., 2007].

- *Cryptanalysis* is a relatively new area of application of SAT which became practical with the recent increase in power of SAT solvers. [Massacci and Marraro, 2000], [Fiorini et al., 2003], [Eibach et al., 2008] describe some of such applications.

We conclude this chapter with a somewhat general observation. Clearly, a good understanding of applications is instrumental for the development of effective application-specific search strategies for SAT. However, in many cases techniques previously developed for the applications turn out to be of great benefit to the development of efficient SAT algorithms as well. A classic example is the techniques of conflict-driven learning and non-chronological backtracking (Section 2.2.1), which were originally developed for the constraint satisfaction problem [Prosser, 1993]. More recently, SAT solvers benefited from the recursive-learning technique developed in ATPG [Marques-Silva and Glass, 1999]. We believe that a study of various applications of SAT is essential to the development of new efficient SAT solving algorithms.

Chapter 4

Conclusion

In conclusion we outline some of the potentially promising research directions that we have alluded to in various parts of the paper.

In Section 3.2, we argued that an efficient incomplete circuit SAT algorithm can be used in some of the industrial applications of SAT, particularly in BMC. Although the development of an such algorithm is a challenging problem, we believe we have identified some of its potential "ingredients", for example:

- Integration of Boolean constraint propagation into the search along the lines of the unit propagation local search algorithm presented in Section 2.3.2.
- Circuit-oriented search heuristics, partly inspired by ATPG research.
- Combination of search with learning techniques, such as the clause learning developed in [Stachniak and Belov, 2008], and some of the learning techniques used in complete circuit SAT solvers (Section 2.6);
- Some ideas from Dynamic Local Search (Section 2.3.1), since DLS algorithms seem to work well on industrial instances [Velev and Bryant, 2001].

Another research direction worth pursuing is the identification of the application areas of SMT which may benefit from the non-clausal SAT algorithms. If non-clausal algorithms turn out to be of use in SMT, the development of efficient incomplete non-clausal algorithms for SAT, along the lines of `polSAT`, (Section 2.5.1) should be pursued. As indicated in Section 3.3, these algorithms may need to be able to search through the space of partial

truth-value assignments – some ideas on how to accomplish this are outlined in [Prestwich, 2000, 2002].

Finally, an issue raised in Section 2.6 should be addressed as well: although ATPG is a well developed area, there seem to be very little information available concerning the applicability of ATPG-specific variable selection heuristics to circuit SAT algorithms.

Bibliography

W. Ackermann. *Solvable Cases of the Decision Problem*. North Holland Publishing Company, 1954.

B. Alpern and F.B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, 1985.

T. Amon, G. Borriello, T. Hu, and J. Liu. Symbolic timing verification of timing diagrams using presburger formulas. In *Proceedings of Design Automation Conference (DAC 1997)*, pages 226–231, 1997.

P.B. Andrews. Theorem proving via general matings. *Journal of ACM*, 28 (2):193–214, 1981.

G. Audemard and L. Simon. GUNSAT: A greedy local search algorithm for unsatisfiability. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI 2007)*, pages 2256–2261, 2007.

T. Ball, C. Cook, S.K. Lahiri, and L. Zhang. Zapato: Automatic theorem proving for predicate abstraction refinement. In *Proceedings of the 16th International Conference on Computer Aided Verification (CAV 2004)*, volume 3114, pages 457–461, 2004.

L. Baptista and J.P. Marques-Silva. Using randomization and learning to solve hard real-world instances of satisfiability. In *Proceedings of the 6th International Conference on Principles and Practice of Constraint Programming (CP 2000)*, pages 489–494, 2000.

C.W. Barrett, D.L. Dill, and A. Stump. Checking satisfiability of first-order formulas by incremental translation to SAT. In *Proceedings of the 14th International Conference on Computer Aided Verification (CAV 2002)*, pages 236–249, 2002.

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 1997)*, pages 203–208, 1997.

P. Beame, H.A. Kautz, and A. Sabharwal. Understanding the power of clause learning. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI 2003)*, pages 1194–1201, 2003.

A. Biere and W. Kunz. SAT and ATPG: Boolean engines for formal hardware verification. In *Proceedings of the 2002 IEEE/ACM International Conference on Computer-Aided Design (ICCAD 2002)*, pages 782–785, 2002.

A. Biere, A. Cimatti, E.M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS 1999)*, pages 193–207, 1999.

P. Bjesse and A. Boralv. DAG-aware circuit compression for formal verification. In *Proceedings of the 2004 IEEE/ACM International Conference on Computer-Aided Design (ICCAD 2004)*, pages 42–49, 2004.

M.P. Bonacina and M. Echenim. Rewrite-based satisfiability procedures for recursive data structures. *Electronic Notes in Theoretical Computer Science*, 174(8):55–70, 2007.

M. Bozzano, R. Bruttomesso, A. Cimatti, A. Franzén, Z. Hanna, Z. Khasidashvili, A. Palti, and R. Sebastiani. Encoding rtl constructs for mathsat: a preliminary report. *Electr. Notes Theor. Comput. Sci.*, 144(2):3–14, 2006.

A. Braunstein, M. Mézard, and R. Zecchina. Survey propagation: An algorithm for satisfiability. *Random Structures and Algorithms*, 27(2):201–226, 2005.

R. Brinkmann and R. Drechsler. Rtl-datapath verification using integer linear programming. In *ASP-DAC '02: Proceedings of the 2002 conference on Asia South Pacific design automation/VLSI Design*, page 741. IEEE Computer Society, 2002.

R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio, Z. Hanna, A. Nadel, A. Palti, and R. Sebastiani. A lazy and layered SMT($\{BV\}$) solver for

hard industrial verification problems. In *Proceedings of the 19th International Conference on Computer Aided Verification (CAV 2007)*, pages 547–560, 2007.

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

R.E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.

J.R. Burch and D.L. Dill. Automatic verification of pipelined microprocessor control. In *Proceedings of the 6th International Conference on Computer Aided Verification (CAV 1994)*, 1994.

J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science (LICS 1990)*, pages 1–33, 1990.

V. Chvátal and E. Szemerédi. Many hard examples for resolution. *Journal of ACM*, 35(4):759–768, 1988.

E.M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Proceedings on Workshop on Logic of Programs*, pages 52–71, 1982.

E.M. Clarke and B.-H. Schlingloff. Model checking. In *Handbook of Automated Reasoning*, volume 2, pages 1635–1790. Elsevier Science Publishers B. V., 2001.

E.M. Clarke, D. Kroening, J. Ouaknine, and O. Strichman. Completeness and complexity of bounded model checking. In *Proceedings of the 5th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI 2004)*, pages 85–96, 2004.

S.A. Cook. The complexity of theorem-proving procedures. In *Conference Record of 3rd Annual ACM Symposium on Theory of Computing (STOC 1971)*, pages 151–158. ACM, 1971.

S. Cotton and O. Maler. Fast and flexible difference constraint propagation for DPLL(T). In *Proceedings of the 9th International Conference on Theory and Applications of Satisfiability Testing (SAT 2006)*, pages 170–183, 2006.

C. Courcoubetis, M. Y. Vardi, P. Wolper, and M. Yannakakis. Memory-efficient algorithms for the verification of temporal properties. *Formal Methods in System Design*, 1(2/3):275–288, 1992.

J.M. Crawford and A.B. Baker. Experimental results on the application of satisfiability algorithms to scheduling problems. In *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI 1994)*, pages 1092–1097, 1994.

E. Dantsin, E.A. Hirsch, S. Ivanov, and M. Vsemirnov. Algorithms for sat and upper bounds on their complexity. *Electronic Colloquium on Computational Complexity (ECCC)*, 8(12), 2001.

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

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

S. Dellacherie, S. Devulder, and J.-L. Lambert. Software verification based on linear programming. In *Proceedings of the Wold Congress on Formal Methods in the Development of Computing Systems (FM 1999)*, pages 1147–1165, 1999.

D. Detlefs, G. Nelson, and J.B. Saxe. Simplify: a theorem prover for program checking. *Journal of ACM*, 52(3):365–473, 2005.

S. Disch and C. Scholl. Combinational equivalence checking using incremental sat solving, output ordering, and resets. In *Proceedings of the 12th Conference on Asia South Pacific Design Automation (ASP-DAC 2007)*, pages 938–943. IEEE, 2007.

N. Eén and N. Sörensson. An extensible SAT-solver. In *Selected Revised Papers of the 6th International Conference on Theory and Applications of Satisfiability Testing (SAT 2003)*, 2004.

N. Eén and N. Sörensson. MiniSat - a SAT solver with conflict-clause minimization. In *Posters of the 8th International Conference on Theory and Applications of Satisfiability Testing (SAT 2005)*, 2005.

T. Eibach, E. Pilz, and G. Völkel. Attacking bivium using SAT solvers. In *Proceedings of the 11th International Conference on Theory and Applications of Satisfiability Testing (SAT 2008)*, pages 63–76, 2008.

F. Fallah, S. Devadas, and K. Keutzer. Functional vector generation for hdl models using linear programming and boolean satisfiability. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 20(8):994–1002, 2001.

H. Fang and M.S. Hsiao. Boosting SAT solver performance via a new hybrid approach. *Journal on Satisfiability, Boolean Modeling and Computation*, 5:243–261, 2008.

H. Fang and W. Ruml. Complete local search for propositional satisfiability. In *Proceedings of the 19th National Conference on Artificial Intelligence (AAAI 2004)*, pages 161–166, 2004.

J. Ferrante and C. Rackoff. A decision procedure for the first order theory of real addition with order. *SIAM Journal on Computing*, 4(1):69–76, 1975.

C. Fiorini, E. Martinelli, and F. Massacci. How to fake an RSA signature by encoding modular root finding as a SAT problem. *Discrete Applied Mathematics*, 130(2):101–127, 2003.

M.J. Fischer and M.O. Rabin. Super-exponential complexity of Presburger arithmetic. In *Proceedings of a Symposium in Applied Mathematics of the American Mathematical Society and the Society for Industrial and Applied Mathematics (SIAMAMS)*, 1974.

C. Flanagan, R. Joshi, X. Ou, and J.B. Saxe. Theorem proving using lazy proof explication. In W.A.Jr. Hunt and F. Somenzi, editors, *Proceedings of the 15th International Conference on Computer Aided Verification (CAV 2003)*, pages 355–367, 2003.

J.W. Freeman. *Improvements to Propositional Satisfiability Search Algorithms*. PhD thesis, University of Pennsylvania, Philadelphia, PA, USA, 1995.

H. Fujiwara and T. Shimono. On the acceleration of test generation algorithms. *IEEE Transactions on Computing*, 32(12):1137–1144, 1983.

M.K. Ganai and A. Kuehlmann. On-the-fly compression of logical circuits. In *Proceedings of International Workshop on Logic Synthesis (IWLS 2000)*, 2000.

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 2002)*, pages 747–750, 2002.

P. Goel. An implicit enumeration algorithm to generate tests for combinational logic circuits. *IEEE Transactions on Computers*, 30(3):215–222, 1981.

E.I. Goldberg, M.R. Prasad, and R.K. Brayton. Using SAT for combinational equivalence checking. In *Proceedings of the Conference on Design,Automation, and Test in Europe (DATE 2001)*, pages 114–121, 2001.

C. Gomes, H. Kautz, A. Sabharwal, and B. Selman. Satisfiability solvers. In F. Van Harmelen, V. Lifschitz, and B. Porter, editors, *Handbook of Knowledge Representation*. Elsevier, Amsterdam, The Netherlands, The Netherlands, 2007.

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

J. Gu. Design efficient local search algorithms. In *Proceedings of the 5th International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems (IEA/AIE 1992)*, pages 651–654, 1992.

J. Gu and R. Puri. Asynchronous circuit synthesis with boolean satisfiability. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 14(8): 961–973, 1995.

G. Gutiérrez, I.P. de Guzmán, J. Martínez, M. Ojeda-Aciego, and A. Valverde. Reduction theorems for boolean formulas using delta-trees. In *Proceedings of the European Workshop on Logics in Artificial Intelligence (JELIA 2000)*, pages 179–192, 2000.

G. Gutiérrez, I.P. de Guzmán, J. Martínez, M. Ojeda-Aciego, and A. Valverde. Satisfiability testing for boolean formulas using delta-trees. *Studia Logica*, 72(1):85–112, 2002.

D. Habet and M. Vasquez. Improving local search for satisfiability problem by integrating structural properties. In *Proceedings of 2007 IEEE International Conference on Research, Innovation and Vision for the Future*, pages 50–57, 2007.

M. Hammer, S. Merz, and I. Lorraine. Truly on-the-fly LTL model checking. In *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005)*, 2005.

I. Hamzaoglu and J. H. Patel. New techniques for deterministic test pattern generation. In *Proceedings of the 16th IEEE VLSI Test Symposium (VTS 1998)*, 1998.

W. Harvey and P. Stukey. A unit two variable per inequality integer constraint solver. In *Proceedings of Australian Computer Science Conference (ACSC 1997)*, pages 102–111, 1997.

M. Heule. *SmArT solving: Tools and techniques for satisfiability solvers*. PhD thesis, Technische Universiteit Delft, 2008.

M. Heule and H. van Maaren. March_dl: Adding adaptive heuristics and a new branching strategy. *Journal on Satisfiability, Boolean Modeling and Computation*, 2:47–59, 2006.

M. Heule and H. van Maaren. march_ks. Solver description submitted to SAT 2007 competition. Available online at: `http://www.satcompetition.org/2007/march_ks.pdf`, 2007.

M. Heule, M. Dufour, J. van Zwieten, and H. van Maaren. March_eq: Implementing additional reasoning into an efficient look-ahead SAT solver. In *Revised Selected Papers of the 7th International Conference Theory and Applications of Satisfiability Testing (SAT 2004)*, pages 345–359, 2004.

E.A. Hirsch and A. Kojevnikov. Unitwalk: A new SAT solver that uses local search guided by unit clause elimination. *Annals of Mathematics and Artificial Intelligence*, 43(1):91–111, 2005.

J. Hoffmann, C.P. Gomes, and B. Selman. Structure and problem hardness: Goal asymmetry and DPLL proofs in SAT-based planning. *Logical Methods in Computer Science*, 3(1), 2007.

G. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1991.

H.H. Hoos. *Stochastic Local Search – Methods, Models, Applications*. PhD thesis, TU Dermstadt, FB Informatik, Darmstadt, Germany, 1998.

H.H. Hoos. On the run-time behaviour of stochastic local search algorithms for SAT. In *Proceedings of the 16th National Conference on Artificial intelligence (AAAI 1999)*, pages 661–666, 1999.

H.H. Hoos and T. Stutzle. Local search algorithms for SAT: An empirical evaluation. *Journal of Automated Reasoning*, 24(4):421–481, 2000.

H.H. Hoos and T. Stutzle. *Stochastic Local Search Foundations and Applications*. Elsevier, 2005.

Holger H. Hoos. An adaptive noise mechanism for walkSAT. In *Proceedings of the 18th National Conference on Artificial intelligence (AAAI 2002)*, pages 655–660, 2002.

F. Hutter, D. Tompkins, and H.H. Hoos. Scaling and probabilistic smoothing: Efficient dynamic local search for SAT. In *Proceedings of the 8th International Conference on Principles and Practice of Constraint Programming (CP 2002)*, pages 233–248, 2002.

G. Istrate. Satisfying assignments of random boolean constraint satisfaction problems: Clusters and overlaps. *Journal of Universal Computer Science*, 13(11):1655–1670, 2007.

H. Jain, C. Bartzis, and E. Clarke. Satisfiability checking of non-clausal formulas using general matings. In *Proceedings of the 9th International Conference on Theory and Applications of Satisfiability Testing (SAT 2006)*, 2006.

M. Järvisalo, T. Junttila, and I. Niemelä. Unrestricted vs restricted cut in a tableau method for Boolean circuits. *Annals of Mathematics and Artificial Intelligence*, 44(4):373–399, 2005.

N. Jussien and O. Lhomme. Local search with constraint propagation and conflict-based heuristics. In *Proceedings of the 17th National Conference on Artificial Intelligence (AAAI 2000)*, pages 169–174, 2000.

H. Kautz, D. McAllester, and B. Selman. Exploiting variable dependency in local search. In *Abstracts of the Poster Sessions of International Joint Conference on Artificial Intelligence (IJCAI 1997)*. 1997.

H.A. Kautz and B. Selman. Planning as satisfiability. In *Proceedings of European Conference on AI*, pages 359–363, 1992.

L.G. Khachiyan. A polynomial algorithm in linear programming. *Soviet Mathematics Doklady*, 20:191–194, 1979.

V. Khomenko, M. Koutny, and A. Yakovlev. Logic synthesis for asynchronous circuits based on STG unfoldings and incremental SAT. *Fundamenta Informaticae*, 70(1):49–73, 2006.

D. Kroening and O. Strichman. Efficient computation of recurrence diameters. In *Proceedings of the 4th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2003)*, pages 298–309, 2003.

A. Kuehlmann. Boolean functions and circuits. Lecture Notes for EECS 219B, Spring 2008, UC Berkely, 2008.

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

O. Kullmann. Investigations on autark assignments. *Discrete Applied Mathematics*, 107(1-3):99–137, 2000.

W. Kunz and D.K. Pradhan. Recursive learning: An attractive alternative to the decision tree for test genration in digital circuits. In *Proceedings of the IEEE International Test Conference on Discover the New World of Test and Design*, pages 816–825, 1992.

W. Kunz and D.K. Pradhan. Accelerated dynamic learning for test pattern generation in combinational circuits. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 12(5):684–694, 1993.

S.K. Lahiri and M. Musuvathi. An efficient decision procedure for utvpi constraints. In *Proceedings of the 5th International Workshop on Frontiers of Combining Systems*, pages 168–183, 2005.

T. Larrabee. Test pattern generation using Boolean satisfiability. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 11(1):4–15, 1992.

T. Latvala, A. Biere, K. Heljanko, and T.A. Junttila. Simple bounded LTL model checking. In *Proceedings of the 5th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2004)*, 2004.

C.-M. Li. Equivalent literal propagation in the DLL procedure. *Discrete Applied Mathematics*, 130(2):251–276, 2003.

C.-M. Li and Anbulagan. Look-ahead versus look-back for satisfiability problems. In *Proceedings of the 3rd International Conference on Principles and Practice of Constraint Programming (CP 1997)*, pages 341–355, 1997a.

C.-M. Li and Anbulagan. Heuristics based on unit propagation for satisfiability problems. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI 1997)*, pages 366–371, 1997b.

F. Lu, L.-C. Wang, K.-T. Cheng, and R. Huang. A circuit SAT solver with signal correlation guided learning. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE 2003)*, 2003.

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

I. Lynce and J.P. Marques-Silva. An overview of backtrack search satisfiability algorithms. *Annals of Mathematics and Artificial Intelligence*, 37 (3):307–326, 2003.

Z. Manna and C.G. Zarba. Combining decision procedures. In *Revised Papers of 10th Anniversary Colloquium of UNU/IIST*, pages 381–422, 2003.

J.P. Marques-Silva and T. Glass. Combinational equivalence checking using satisfiability and recursive learning. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE 1999)*, pages 145–149. IEEE Computer Society, 1999.

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 1996)*, pages 220–227, 1996.

J.P. Marques-Silva and K. A. Sakallah. The impact of branching heuristics in propositional satisfiability algorithms. In *Proceedings of the 9th Portuguese Conference on Artificial Intelligence (EPIA 1999)*, pages 62–74, 1999.

J.P. Marques-Silva and K.A. Sakallah. Robust search algorithms for test pattern generation. In *Proceedings of the International Symposium on Fault-Tolerant Computing (FTCS 1997)*, pages 152–161, 1997.

F. Massacci and L. Marraro. Logical cryptanalysis as a SAT problem. *Journal of Automated Reasoning*, 24(1/2):165–203, 2000.

B. Mazure, L. Sais, and E. Grégoire. Tabu search for SAT. In *Proceedings of the 14th National Conference on Artificial Intelligence (AAAI 1997)*, pages 281–285, 1997.

D. McAllester, B. Selman, and H. Kautz. Evidence for invariants in local search. In *Proceedings of the 14th National Conference on Artificial Intelligence (AAAI 1997)*, pages 321–326, 1997.

D.A. McAllester. An outlook on truth maintenance. AI Memo 551, Artificial Intelligence Laboratory, MIT, 1980.

D.A. McAllester. Truth maintenance. In *Proceedings of the 8th National Conference on Artificial Intelligence (AAAI 1990)*, pages 1109–1116, 1990.

S.O. Memik and F. Fallah. Accelerated SAT-based scheduling of control/data flow graphs. In *Proceedings of the 20th International Conference on Computer Design (ICCD 2002)*, 2002.

M. Mezard, G. Parisi, and R. Zecchina. Analytic and algorithmic solution of random satisfiability problems. *Science*, 297(5582):812–815, 2002.

D.G. Mitchell. A SAT solver primer. *Bulletin of the EATCS*, 85, 2005.

M.M. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: engineering an efficient SAT solver. In *Proceedings of the 38th Conference on Design Automation (DAC 2001)*, pages 530–535, 2001.

R. Muhammad and P.J. Stuckey. A stochastic Non-CNF SAT solver. In *Proceedings of the 9th Pacific Rim International Conference on Artificial Intelligence (PRICAI 2006)*, 2006.

G. Nelson. Techniques for program verification. Technical Report CSL-81-10, Xerox Palo Alto Reasearch Center, 1981.

R. Nieuwenhuis and A. Oliveras. Proof-producing congruence closure. In *Proceedings of the 16th International Conference on Term Rewriting and Applications (RTA 2005)*, 2005.

D.C. Oppen. Elementary bounds for Presburger arithmetic. In *Proceedings of the 5th annual ACM Symposium on Theory of Computing (STOC 1973)*, 1973.

C.H. Papadimitriou. On the complexity of integer programming. *Journal of ACM*, 28(4):765–768, 1981.

R. Paturi, P. Pudlak, and F. Zane. Satisfiability coding lemma. In *Proceedings of the 38th Annual Symposium on Foundations of Computer Science (FOCS 1997)*, page 566, 1997.

R. Paturi, P. Pudlák, M.E. Saks, and F. Zane. An improved exponential-time algorithm for k-SAT. In *Proceedings of the 39th Annual Symposium on Foundations of Computer Science (FOCS 1998)*, page 628, 1998.

D.N. Pham and C. Gretton. gnovelty+. Solver description submitted to SAT 2007 competition. Available online at: `http://www.satcompetition.org/2007/gnovelty+.pdf`, 2007.

V. Pratt. Two easy theories whose combination is hard. Technical report, Massachusetts Institute of Technology, 1977.

M. Presburger. Über die Vollstäendigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt. In *Comptes-rendus du I Congrés des Mathématiciens des Pays Slaves, Varsovie 1929*, pages 92–101,395, 1930.

S.D. Prestwich. A hybrid search architecture applied to hard random 3-SAT and low-autocorrelation binary sequences. In *Proceedings of the 6th International Conference on Principles and Practice of Constraint Programming (CP 2000)*, pages 337–352, 2000.

S.D. Prestwich. Randomised backtracking for linear pseudo-boolean constrainty problems. In *In Proceedings of the 4th International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimisation Problems (CPAIOR 2002)*, 2002.

S.D. Prestwich and I. Lynce. Local search for unsatisfiability. In *Proceedings of the 9th International Conference of Theory and Applications of Satisfiability Testing (SAT 2006)*, 2006.

D. Pretolani. Efficiency and stability of hypergraph SAT algorithms. In *Proceedings of DIMACS Challenge II Workshop*, 1993.

P. Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9(3):268–299, 1993.

W. Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 4–13, 1991.

J.-P. Queille and J. Sifakis. Specification and verification of concurrent systems in cesar. In *Proceedings of the 5th Colloquium on International Symposium on Programming*, pages 337–351, 1982.

E.T. Richards and B. Richards. Nonsystematic search and no-good learning. *Journal of Automated Reasoning*, 24(4):483–533, 2000.

J. Rintanen, K. Heljanko, and I. Niemelä. Planning as satisfiability: parallel plans and algorithms for plan search. *Artificial Intelligence*, 170(12-13): 1031–1080, 2006.

S. Safarpour, A. Veneris, G. Baeckler, and R. Yuan. Efficient SAT-based Boolean matching for FPGA technology mapping. In *Proceedings of the 43rd Annual Conference on Design Automation (DAC 2006)*, pages 466–471, 2006.

M.H. Schulz, E. Trischler, and T.M. Sarfert. SOCRATES: a highly efficient automatic test pattern generation system. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 7(1):126–137, 1988.

D. Schuurmans, F. Southey, and R.C. Holte. The exponentiated subgradient algorithm for heuristic boolean programming. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI 2001)*, pages 334–341, 2001.

R. Sebastiani. Lazy satisfiability modulo theories. *Journal on Satisfiability, Boolean Modeling and Computation*, 1, 2006.

R. Sebastiani. Applying GSAT to non-clausal formulas (research note). *Journal of Artificial Intelligence Research (JAIR)*, 1:309–314, 1994.

B. Selman and H.A. Kautz. Domain-independent extensions to GSAT: Solving large structured satisfiability problems. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI 1993)*, pages 290–295, 1993.

B. Selman, H.J. Levesque, and D.G. Mitchell. A new method for solving hard satisfiability problems. In *Proceedings of the 10th National Conference on Artificial Intelligence (AAAI 1992)*, pages 440–446, 1992.

B. Selman, H.A. Kautz, and B. Cohen. Noise strategies for improving local search. In *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI 1994)*, pages 337–343, 1994.

Z. Stachniak. Going non-clausal. In *Proceedings of the 5th International Symposium on the Theory and Applications of Satisfiability Testing (SAT 2002)*, 2002.

Z. Stachniak. Polarity guided tractable reasoning. In *Proceedings of the 17th National Conference on Artificial intelligence (AAAI 1999)*, pages 751–758, 1999.

Z. Stachniak and A. Belov. Speeding-up non-clausal local search for propositional satisfiability with clause learning. In *Proceedings of the 11th International Conference on Theory and Applications of Satisfiability Testing (SAT 2008)*, pages 257–270, 2008.

A. Stump, C.W. Barrett, D.L. Dill, and J.R. Levitt. A decision procedure for an extensional theory of arrays. In *Proceedings of IEEE Symposium on Logic in Computer Science (LICS 2001)*, pages 29–37, 2001.

A. Tarski. *A decision method for elementary algebra and geometry.* The Rand Corporation, Santa Monica, California, 1948.

C. Thiffault, F. Bacchus, and T. Walsh. Solving non-clausal formulas with DPLL search. In *Proceedings of the 10th International Conference on Principles and Practice of Constraint Programming (CP 2004)*, pages 663–678, 2004.

URL-a. SAT competition website. `http://www.satcompetition.org`.

URL-b. SatMate website. `http://www.cs.cmu.edu/ modelcheck/satmate/`.

A. Van Gelder. A satisfiability tester for non-clausal propositional calculus. *Information and Computation*, 79(1):1–21, 1988.

M.Y. Vardi. Branching vs. linear time: Final showdown. In *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2001)*, pages 1–22, 2001.

M.N. Velev and R.E. Bryant. Effective use of boolean satisfiability procedures in the formal verification of superscalar and vliw microprocessors. In *Proceedings of the 38th Design Automation Conference (DAC 2001)*, pages 226–231, 2001.

D. Wagner, J.S. Foster, E.A. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Proceedings of the Network and Distributed System Security Symposium (NDSS 2000)*, 2000.

R.G. Wood and R.A. Rutenbar. FPGA routing and routability estimation via Boolean satisfiability. *IEEE Transactions on VLSI Systems*, 6(2): 222–231, 1998.

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

L. Xu, H. Hutter, H.H. Hoos, and K. Leyton-Brown. SATZilla: Portfolio-based algorithm selection for SAT. *Journal of Artificial Intelligence Research*, 32:565–606, 2008.

H. Zhang, D. Li, and H. Shen. A SAT based scheduler for tournament schedules. In *Proceedings of the 7th International Conference on Theory and Applications of Satisfiability Testing (SAT 2004)*, 2004.