Boolean Satisfiability Solvers: Techniques and Extensions

Georg WEISSENBACHER [a] and Sharad MALIK [a]
[a] *Princeton University*

Abstract. Contemporary satisfiability solvers are the corner-stone of many successful applications in domains such as automated verification and artificial intelligence. The impressive advances of SAT solvers, achieved by clever engineering and sophisticated algorithms, enable us to tackle *Boolean Satisfiability* (SAT) problem instances with millions of variables – which was previously conceived as a hopeless problem. We provide an introduction to contemporary SAT-solving algorithms, covering the fundamental techniques that made this revolution possible. Further, we present a number of extensions of the SAT problem, such as the enumeration of *all satisfying assignments* (ALL-SAT) and determining the *maximum number of clauses that can be satisfied by an assignment* (MAX-SAT). We demonstrate how SAT solvers can be leveraged to solve these problems. We conclude the chapter with an overview of applications of SAT solvers and their extensions in automated verification.

Keywords. Satisfiability solving, Propositional logic, Automated decision procedures

1. Introduction

Boolean Satisfibility (SAT) is the problem of checking if a propositional logic formula can ever evaluate to true. This problem has long enjoyed a special status in computer science. On the theoretical side, it was the first problem to be classified as being NP-complete. NP-complete problems are notorious for being hard to solve; in particular, in the worst case, the computation time of any known solution for a problem in this class increases exponentially with the size of the problem instance. On the practical side, SAT manifests itself in several important application domains such as the design and verification of hardware and software systems, as well as applications in artificial intelligence. Thus, there is strong motivation to develop practically useful SAT solvers.

However, the NP-completeness is cause for pessimism, since it is unlikely that we will be able to scale the solutions to large practical instances. While attempts to develop practically useful SAT solvers have persisted for almost half a century, for the longest time it was a largely academic exercise with little hope of seeing practical use. Fortunately, several relatively recent research developments have enabled us to tackle instances with millions of variables and constraints – enabling SAT solvers to be effectively deployed in practical applications including in the analysis and verification of software.

This chapter provides an introduction to contemporary SAT-solving techniques and is organised as follows: Section 2 introduces the syntax and semantics of propositional logic. The remaining chapter is split into three parts: The first part (Section 3) covers the techniques used in modern SAT solvers. Further, it covers basic extensions such as the constructions of unsatisfiability proofs. For instances that are unsatisfiable, the proofs of unsatisfiability have been used to derive an unsatisfiable subset of constraints of the formula, referred to as the UNSAT core. The UNSAT core has seen successful applications in model checking. The second part (Section 4) considers extensions of these solvers that have proved to be useful in analysis and verification. Related to the UNSAT core are the concepts of minimal correction sets and maximally satisfiable subsets. A maximally satisfiable subset of an unsatisfiable instance is a maximal subset of constraints that is satisfiable, and a minimal correction set is a minimal subset of constraints that needs to be dropped to make the formula satisfiable. Section 4 discusses how these concepts are related and covers algorithms to derive them. The third part (Section 5) discusses applications of the techniques presented in the Sections 3 and 4 in the field of automated verification. These applications include automated test case generation, bounded model checking and equivalence checking, and fault localisation. Finally, Appendix A provides a number of exercises and their solutions.

2. Preliminaries

This section establishes the notation and syntax we employ throughout this chapter and the meaning (semantics) that is assigned to it.

2.1. Propositional Logic

Propositional logic is a formalism that enables us to make statements about *propositions* (or variables). While propositions may have some underlying meaning associated with them (e.g., the implicit meaning of x_1 being true may be that "it is raining outside"), we do not concern ourselves with such interpretations, but merely require that each proposition can have exactly one of two truth values (true or false).

2.1.1. Notation

Let $\mathcal{V}$ be a set of n propositional logic variables and let $\mathsf{0}$ and $\mathsf{1}$ denote the elements of the Boolean domain $\mathbb{B}$ representing false and true, respectively. Every Boolean function f $\mathbb{B}^n \longrightarrow \mathbb{B}$ can be expressed as a propositional logic formula F in n variables $x_1, \ldots, x_n \in \mathcal{V}$. The syntax of propositional logic formulae is provided in Figure 1.

The interpretation of the logical connectives $\{-, +, \cdot, \rightarrow, \leftrightarrow, \oplus\}$ is provided in Table 1. We use $\equiv$ to denote logical equivalence. For brevity, we may omit $\cdot$ in conjunctions (e.g., $x_1\overline{x}_3$). An assignment $\mathcal{A}$ is a mapping from $\mathcal{V}$ to $\mathbb{B}$, and $\mathcal{A}(x)$ denotes the value that $\mathcal{A}$ assigns to x. We call $\mathcal{A}$ a *total* assignment if $\mathcal{A}$ is a total function. Otherwise, $\mathcal{A}$ is a partial assignment. $\mathcal{A}$ *satisfies* a formula $F(x_1, \ldots x_n)$ iff $F(\mathcal{A}(x_1), \ldots, \mathcal{A}(x_n))$ is defined and evaluates to $\mathsf{1}$ (denoted by $\mathcal{A} \models F$). A formula F is satisfiable iff $\exists \mathcal{A} \,.\, \mathcal{A} \models F$, and unsatisfiable (inconsistent, respectively) otherwise. We use $\#\mathcal{A}_F$ to denote the number of satisfying total assignments of a formula F and drop the subscript if F is clear from the context. A formula F *holds* iff $\mathcal{A} \models F$ for all total assignments $\mathcal{A}$.

We use $\mathtt{Lit}_{\mathcal{V}} = \{x, \overline{x} \mid x \in \mathcal{V}\}$ to denote the set of literals over $\mathcal{V}$, where $\overline{x}$ is the negation of x. Given a literal $\ell \in \mathtt{Lit}_{\mathcal{V}}$, we write $\mathrm{var}(\ell)$ to denote the variable occuring in ℓ. A *cube* over $\mathcal{V}$ is a product of literals $\ell_1 \ldots \ell_m$ such that $\ell_i \in \mathtt{Lit}_{\mathcal{V}}$ and

x	y	$\overline{x}$	$x \cdot y$	$x + y$	$x \rightarrow y$	$x \leftrightarrow y$	$x \oplus y$
0	0	1	0	0	1	1	0
0	1	1	0	1	1	0	1
1	0	0	0	1	0	0	1
1	1	0	1	1	1	1	0

Table 1. Definition of Propositional Logic Operators

$$\begin{array}{rcl} \mathit{formula} & ::= & \mathit{formula} \cdot \mathit{formula} \mid \mathit{formula} + \mathit{formula} \mid \\ & & \mathit{formula} \rightarrow \mathit{formula} \mid \mathit{formula} \leftrightarrow \mathit{formula} \mid \\ & & \mathit{formula} \oplus \mathit{formula} \mid \overline{\mathit{formula}} \mid (\mathit{formula}) \mid \mathit{atom} \\ \mathit{atom} & ::= & \mathit{propositional\ identifier} \mid \mathit{constant} \\ \mathit{constant} & ::= & \mathsf{1} \mid \mathsf{0} \end{array}$$

Figure 1. Syntax of Propositional Logic

$$\begin{array}{rcl} \mathit{formula} & ::= & \mathit{formula} \cdot (\mathit{clause}) \mid (\mathit{clause}) \\ \mathit{clause} & ::= & \mathit{clause} + \mathit{literal} \mid \mathit{literal} \\ \mathit{literal} & ::= & \mathit{atom} \mid \overline{\mathit{atom}} \\ \mathit{atom} & ::= & \mathit{propositional\ identifier} \end{array}$$

Figure 2. Syntax of Propositional Logic in Conjunctive Normal Form

$\mathrm{var}(\ell_i) \neq \mathrm{var}(\ell_j)$ for all $i, j \in \{1..m\}$ with $i \neq j$. We write $\ell \in C$ to indicate that the literal ℓ occurs in a cube C. Given an assignment $\mathcal{A}$, we use $C_{\mathcal{A}}$ to denote the cube $\prod_{i=1}^{n} \ell_i$ where $\ell_i = x_i$ if $\mathcal{A}(x_i) = \mathsf{1}$ and $\ell_i = \overline{x}_i$ otherwise.

2.1.2. Conjunctive Normal Form

The Conjunctive Normal Form (CNF) of a formula is a restricted form of propositional logic formulae. Figure 2 shows the syntax of propositional logic formulae in CNF. A formula in CNF is product of sums (a conjunction of clauses, respectively). Note that the empty clause (denoted by $\Box$) corresponds to the logical value $\mathsf{0}$.

The formula $\overline{x}_1 \cdot (x_1 + \overline{x}_2) \cdot (\overline{x}_1 + x_2) \cdot x_1$ is in CNF, for instance. A commonly used alternative (and more compact) representation of this formula is $(\overline{x}_1)$ $(x_1\, \overline{x}_2)$ $(\overline{x}_1\, x_2)$ (x_1), i.e., the logical connectives $\cdot$ and $+$ are dropped, since they are clear from the context.

Clauses are commonly regarded as *sets of literals*. While we stick to the notation defined in Figure 2, we will implicitly assume that clauses have the properties of sets of literals. Accordingly, $(\overline{x}_1\, x_2\, x_2)$ and $(x_1\, x_2)$ are indistinguishable from their logically equivalent counterparts $(\overline{x}_1\, x_2)$ and $(x_2\, x_1)$, respectively. Therefore, a formula in CNF is a set of sets of literals. Note that this representation implicitly incorporates *factoring* (i.e., merging of unifiable literals).

Each formula F in propositional logic can be transformed into CNF. Unfortunately, the resulting formula may be exponentially larger than F. It is, however, possible to construct a formula G in CNF such that F and G are *equi-satisfiable* (i.e., $(\exists \mathcal{A} \,.\, \mathcal{A} \models F) \leftrightarrow (\exists \mathcal{A} \,.\, \mathcal{A} \models G)$) and the size of G is polynomial in the size of the original formula F. Such an equi-satisfiable formula can be obtained by means of Tseitin's transformation [Tse83]. Given a formula F in propositional logic (as defined in Figure 1), this transformation involves the following steps:

1. Recursively replace each sub-formula $(F_1 \rhd F_2)$ of the original formula F (where $\rhd \in \{-,+,\cdot,\rightarrow,\leftrightarrow,\oplus\}$) with a fresh propositional identifier x and add the constraint $x \leftrightarrow (F_1 \rhd F_2)$.
2. Rewrite the resulting formula into CNF by using the rules presented in Table 2.

Example 2.1 *We demonstrate Tseitin's transformation by converting the formula* $\overline{(y \leftrightarrow z)}$ *into conjunctive normal form.*

1. *The first step is to replace* $(y \leftrightarrow z)$ *with a fresh propositional identifier* x_1*. After adding the corresponding constraint, we obtain* $\overline{x}_1 \cdot (x_1 \leftrightarrow (y \leftrightarrow z))$
2. *In the next step, we replace* $\overline{x}_1$ *with* x_2*. This step is optional, since* $(\overline{x}_1)$ *is already in clausal form. This transformation step yields the formula*

$$x_2 \cdot (x_2 \leftrightarrow \overline{x}_1) \cdot (x_1 \leftrightarrow (y \leftrightarrow z)) \ .$$

3. *This formula can be rewritten according to Table 2:*

$$x_2 \cdot \underbrace{(x_2 \leftrightarrow \overline{x}_1)}_{(\overline{x_1}+\overline{x}_2)\cdot(x_1+x_2)} \cdot \underbrace{(x_1 \leftrightarrow (y \leftrightarrow z))}_{(\overline{x}_1+\overline{y}+z)\cdot(\overline{x}_1+\overline{z}+y)\cdot(\overline{y}+\overline{z}+x_1)\cdot(y+z+x_1)}$$

4. *We obtain an equi-satisfiable formula in CNF:*

$$x_2 \cdot (\overline{x_1}+\overline{x}_2) \cdot (x_1+x_2) \cdot (\overline{x}_1+\overline{y}+z) \cdot (\overline{x}_1+\overline{z}+y) \cdot (\overline{y}+\overline{z}+x_1) \cdot (y+z+x_1)$$

We also encourage the reader to solve the Exercises 1 and 2 in Section A.

3. Boolean Satisfiability Checking: Techniques

In this section, we formally introduce the problem of Boolean satisfiability (SAT) and present a number of techniques to tackle it.

3.1. Problem Definition

Definition 3.1 (Boolean Satisfiability Problem) *Given a propositional logic formula* F*, determine whether* F *is satisfiable.*

The Boolean Satisfiability Problem, usually referred to as SAT, is a prototypical NP-complete problem [Coo71], i.e., there is no known algorithm that efficiently solves all instances of SAT. While Definition 3.1 refers to formulae in propositional logic in general, the problem can be easily reduced to formulae in CNF: Using Tseitin's transformation (c.f. Section 2.1.2), any arbitrary propositional formula can be transformed into an equi-satisfiable formula in clausal form. It is therefore sufficient to focus on formulae in conjunctive normal form.

Negation:

$$\begin{aligned} x \leftrightarrow \overline{y} &\equiv (x \rightarrow \overline{y}) \cdot (\overline{y} \rightarrow x) \\ &\equiv (\overline{x} + \overline{y}) \cdot (y + x) \end{aligned}$$

Disjunction:

$$\begin{aligned} x \leftrightarrow (y + z) &\equiv (y \rightarrow x) \cdot (z \rightarrow x) \cdot (x \rightarrow (y + z)) \\ &\equiv (\overline{y} + x) \cdot (\overline{z} + x) \cdot (\overline{x} + y + z) \end{aligned}$$

Conjunction:

$$\begin{aligned} x \leftrightarrow (y \cdot z) &\equiv (x \rightarrow y) \cdot (x \rightarrow z) \cdot ((y \cdot z) \rightarrow x) \\ &\equiv (\overline{x} + y) \cdot (\overline{x} + z) \cdot (\overline{(y \cdot z)} + x) \\ &\equiv (\overline{x} + y) \cdot (\overline{x} + z) \cdot (\overline{y} + \overline{z} + x) \end{aligned}$$

Equivalence:

$$\begin{aligned} x \leftrightarrow (y \leftrightarrow z) &\equiv (x \rightarrow (y \leftrightarrow z)) \cdot ((y \leftrightarrow z) \rightarrow x) \\ &\equiv (x \rightarrow ((y \rightarrow z) \cdot (z \rightarrow y)) \cdot ((y \leftrightarrow z) \rightarrow x) \\ &\equiv (x \rightarrow (y \rightarrow z)) \cdot (x \rightarrow (z \rightarrow y)) \cdot ((y \leftrightarrow z) \rightarrow x) \\ &\equiv (\overline{x} + \overline{y} + z) \cdot (\overline{x} + \overline{z} + y) \cdot ((y \leftrightarrow z) \rightarrow x) \\ &\equiv (\overline{x} + \overline{y} + z) \cdot (\overline{x} + \overline{z} + y) \cdot (((y \cdot z) + (\overline{y} \cdot \overline{z})) \rightarrow x) \\ &\equiv (\overline{x} + \overline{y} + z) \cdot (\overline{x} + \overline{z} + y) \cdot ((y \cdot z) \rightarrow x) \cdot ((\overline{y} \cdot \overline{z}) \rightarrow x) \\ &\equiv (\overline{x} + \overline{y} + z) \cdot (\overline{x} + \overline{z} + y) \cdot (\overline{y} + \overline{z} + x) \cdot (y + z + x) \end{aligned}$$

Table 2. Tseitin transformation [Tse83] for standard Boolean connectives

There are two important sub-classes of SAT:

- *2-SAT.* Each clause of the formula contains at most 2 literals. The satisfiability of such 2-CNF formulae can be decided in polynomial time [Kro67]: each clause $(\ell_1\,\ell_2)$ can be rewritten as an implication $\overline{\ell}_1 \rightarrow \ell_2$ (or $\mathsf{1} \rightarrow \ell_1$ and $\overline{\ell}_1 \rightarrow \mathsf{0}$ in case of a clause (ℓ_1) with only one literal). The formula is satisfiable if the transitive closure of the implications does not yield $\mathsf{0}$. This approach effectively amounts to resolution (see Section 3.2).
- *3-SAT.* Each clause of the formula contains at most 3 literals. This form is relevant because any arbitrary formula in CNF can be reduced to an equi-satisfiable 3-CNF formula by means of Tseitin's transformation (Section 2.1.2).

3.2. *Resolution Proofs*

The *resolution principle* states that an assignment satisfying the clauses $C + x$ and $D + \overline{x}$ also satisfies $C + D$. The clauses $C + x$ and $D + \overline{x}$ are the *antecedents*, x is the *pivot*, and $C + D$ is the *resolvent*. Let $\text{Res}(C, D, x)$ denote the resolvent of the clauses C and D with the pivot x. The corresponding resolution rule is formally described below.

$$\frac{C + x \qquad D + \overline{x}}{C + D}\ \mathsf{Res}$$

Resolution corresponds to existential quantification of the pivot and subsequent quantifier elimination, as demonstrated by the following sequence of logical transformation steps (where $F_{(x \leftarrow e)}$ denotes the substitution of all free occurrences of x in F with the expression e):

$$\begin{aligned}
&\exists x\,.\,(C+x)\;\cdot\;(D+\overline{x})\\
&\equiv ((C+x)\cdot(D+\overline{x}))_{(x\leftarrow 1)} + ((C+x)\cdot(D+\overline{x}))_{(x\leftarrow 0)}\\
&\equiv \underbrace{(C+1)}_{1}\cdot\underbrace{(D+\overline{1})}_{D}+\underbrace{(C+0)}_{C}\cdot\underbrace{(D+\overline{0})}_{1}\\
&\equiv C+D
\end{aligned}$$

The repeated application of the resolution rule results in a resolution proof.

Definition 3.2 *A* resolution proof *R is a directed acyclic graph $(V_R, E_R, piv_R, \lambda_R, \mathsf{s}_R)$, where V_R is a set of vertices, E_R is a set of edges, piv_R is a pivot function, λ_R is the clause function, and $\mathsf{s}_R \in V_R$ is the sink vertex. An* initial vertex *has in-degree* 0*. All other vertices are* internal *and have in-degree* 2*. The sink has out-degree* 0*. The pivot function maps internal vertices to pivot variables of the respective resolution step. For each internal vertex v and $(v_1, v), (v_2, v) \in E_R$, $\lambda_R(v) = \mathrm{Res}(\lambda_R(v_1), \lambda_R(v_2), piv_R(v))$.*

A resolution proof R is a refutation if $\lambda_R(\mathsf{s}_R) = \square$. A refutation R is a refutation for a formula F (in CNF) if the label of each initial vertex of R is a clause of F.

Example 3.1 (Unit Propagation and Resolution) *Figure 3 shows an example of a resolution proof for the formula*

$$(\overline{x}_1)\cdot(x_1+\overline{x}_2)\cdot(\overline{x}_1+x_2)\cdot(x_1)\,. \tag{1}$$

In Figure 3, each node v is represented by its label $\lambda(v)$ (the parentheses around the literals are dropped since each node is associated with exactly one clause and there is no risk of ambiguity). Moreover, Figure 3 does not show the pivot variables explicitly, since they are uniquely determined by the clauses labelling a node and its predecessors and therefore clear from the context.

Note that this formula is a 2-CNF formula and can therefore be solved by means of transitive closure of the corresponding implications. Equivalently, the unsatisfiability of Formula (1) can be established by repeated application of the unit-resolution rule*:*

$$\frac{\ell \qquad D+\overline{\ell}}{D}\;\textsf{URes}$$

Here, ℓ denotes a literal over the pivot variable.

3.3. The Davis-Putnam Procedure

The resolution rule is sufficient to devise a complete algorithm for deciding the satisfiability of a CNF formula [Rob65].

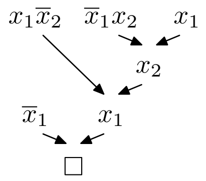


Figure 3. Resolution proof

Theorem 3.1 (Completeness of Propositional Resolution) *If F is an inconsistent formula in CNF, then there is a resolution refutation for F.*

Proof sketch. By induction over the number of variables in F (c.f. [Bus98]). In the base case, where no variables appear in F, the formula must contain the empty clause $\Box$. For the induction step, let x be a fixed variable in F, and let F_1 to be the formula defined as follows:

1. For all clauses $(C+x)$ and $(D+\overline{x})$ in F, the resolvent $\text{Res}((C+x),\,(D+\overline{x}),\,x)$ is in F_1.
2. Every clause C in F which contains neither x nor $\overline{x}$ is in F_1.

It is clear that x does not occur in F_1 unless F contains trivial clauses C for which $\{x,\overline{x}\} \subseteq C$. W.l.o.g., such tautological clauses can be dropped. Then, F_1 is satisfiable if and only if F is, from whence the theorem follows by the induction hypothesis.

Remark Resolution is merely *refutation-complete*, i.e., while it is always possible to derive $\Box$ from an inconsistent formula, it does not enable us to derive all valid implications: we cannot deduce $(x + y)$ from (x) by means of resolution, for instance, even though the latter obviously entails the former.

The constructive proof sketch above is interesting for two reasons:

- It demonstrates that propositional resolution is refutation-complete even if we fix the order of pivots along each path in the proof, and
- it outlines a decision procedure which is known as Davis-Putnam procedure [DP60].

We refer to the algorithm presented in [DP60] as "Davis-Putnam" procedure or simply DP. The Davis-Putnam procedure comprises three rules:

1. *1-literal rule.* Whenever one of the clauses in F is a *unit clause*, i.e., contains only a single literal ℓ, then we obtain a new formula F_1 by

   (a) removing any instances of $\overline{\ell}$ from the other clauses, and
   (b) removing any clause containing ℓ, including the unit clause itself.

   This rule obviously subsumes unit-resolution (see Example 3.1).
2. *The affirmative-negative rule.* If any literal ℓ occurs *only positively* or *only negatively* in F, then remove all clauses containing ℓ. This transformation obviously preserves satisfiability.
3. *The rule for eliminating atomic formulae.* For all clauses $(C+x)$ and $(D+\overline{x})$ in F, where neither C nor D contain x or $\overline{x}$, the resolvent $\text{Res}((C+x),\,(D+\overline{x}),\,x)$ is in F_1. Moreover, every clause C in F which contains neither x nor $\overline{x}$ is in F_1.

The last rule can make the formula increase in size significantly. However, it completely eliminates all occurrences of the atom x. The correctness of the transformation is justified by the resolution principle (see Section 3.2).

In practice, the resolution rule should only be applied *after* the 1-literal rule and affirmative-negative rule. The 1-literal rule is also known as *unit propagation* and lends itself to efficient implementations.

Once this option is exhausted, we face a choice of which pivot variable x to resolve on. While there is no "wrong" choice that forfeits completeness (as established in the proof of Theorem 3.1), a "bad" choice of a pivot may result in a significant blowup of the formula, and therefore retard the performance of the solver. We postpone the discussion of selection strategies to Section 3.7.

3.4. The Davis-Putnam-Logeman-Loveland Procedure

For realistic problems, the number of clauses generated by the DP procedure grows quickly. To avoid this explosion, Davis, Logemann, and Loveland [DLL62] suggested to replace the resolution rule with a case split. This modified algorithm is commonly referred to as DPLL procedure. It is based on the identity known as Shannon's expansion [Sha49]:

$$F \equiv x \cdot F_{(x \leftarrow 1)} + \overline{x} \cdot F_{(x \leftarrow 0)} \tag{2}$$

Accordingly, checking the satisfiability of a formula F can be reduced to testing $F \cdot x$ and $F \cdot \overline{x}$ separately. The subsequent application of unit propagation (the 1-literal rule, respectively) can reduce the size of these formulae significantly. This transformation, applied recursively, yields a complete decision procedure.

In practice, this split is not implemented by means of recursion but in an iterative manner (using tail recursion, respectively). We keep track of the recursive case-splits and their implications using an explicit trail. Each entry in this trail represents an assignment to a variable of F imposed by either a case split or by unit propagation. We refer to the former kind of entries as *guessed* and to the latter as *implied* assignments.

Definition 3.3 (Clauses under Partial Assignments) *A trail represents a* partial *assignment $\mathcal{A}$ to the variables $\mathcal{V}$ of F.*

- *A clause C is* satisfied *if one or more of its literals evaluates to* 1 *under the partial assignment $\mathcal{A}$.*
- *A clause C is* conflicting *if all of its literals are assigned and C evaluates to* 0 *under $\mathcal{A}$.*
- *A clause C becomes* unit *under a partial assignment if all but one of its literals are assigned but C is not satisfied. As such, C gives rise to an implied assignment. In this case, we say that C is the* antecedent *of the implied assignment.*
- *In all other cases, we say that the clause C is* unresolved.

Example 3.2 *Consider the clauses*

$$C_1 \equiv (\overline{x}_1\, \overline{x}_4\, x_3) \qquad \textit{and} \qquad C_2 \equiv (\overline{x}_3\, \overline{x}_2)\,.$$

Level	**Partial Assignment**	**Clauses**	**Trail**
0	–	$(\overline{x}_1\,\overline{x}_4\,x_3)(\overline{x}_3\overline{x}_2)$	
1	$\{x_1 \mapsto 1\}$	$(\overline{x}_1\,\overline{x}_4\,x_3)(\overline{x}_3\overline{x}_2)$	x_1, guessed
2	$\{x_1 \mapsto 1, x_4 \mapsto 1\}$	$(\overline{x}_1\,\overline{x}_4\,x_3)(\overline{x}_3\overline{x}_2)$	x_4, guessed
	$\{x_1 \mapsto 1, x_4 \mapsto 1, x_3 \mapsto 1\}$	$(x_3)(\overline{x}_3\overline{x}_2)$	x_3, implied
	$\{x_1 \mapsto 1, x_4 \mapsto 1, x_3 \mapsto 1, x_2 \mapsto 0\}$	$(\overline{x}_2)$	$\overline{x}_2$, implied

Table 3. Assignment trail for Example 3.2

Table 3 shows a possible trail for this instance. Initially, neither of the clauses is unit, forcing us to guess *an assignment for one of the variables and thus to introduce a new decision. We choose to explore the branch in which x_1 is assigned 1 first. The first entry in the trail, the literal x_1, represents this decision. Neither of the clauses is unit under this assignment; we decide to assign x_4. The clause C_1 is unit under the partial assignment $\{x_1 \mapsto 1, x_4 \mapsto 1\}$ and implies the assignment $x_3 \mapsto 1$ (note that we mark the assignment as "implied" in the trail). This assignment, in turn, makes C_2 unit, imposing the assignment $x_2 \mapsto 0$. The resulting assignment satisfies C_1 as well as C_2.*

A trail may lead to a dead end, i.e., result in a conflicting clause, in which case we have to explore the alternative branch of one of the case splits previously made. This corresponds to reverting one of the decisions or *backtracking*, respectively.

Example 3.3 (Backtracking) *Consider the set of clauses*

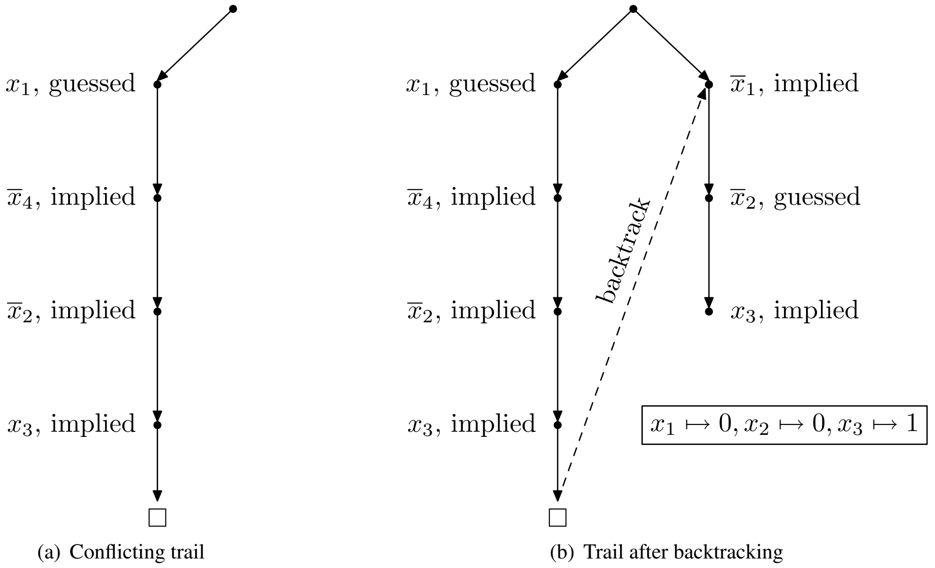
$$C_1 \equiv (x_2\,x_3) \quad C_2 \equiv (\overline{x}_1\overline{x}_4) \quad C_3 \equiv (\overline{x}_2 x_4) \quad C_4 \equiv (\overline{x}_1 x_2 \overline{x}_3)\,.$$

Figure 4(a) shows a trail that leads to a conflict (assignments are represented as literals, c.f. Section 2.1.1). Clause C_4 is conflicting under the given assignment. The last (and only) guessed assignment on the given trail is $x_1 \mapsto 1$. Accordingly, we backtrack *to this most recent decision (dropping all implications made after this point) and revert it to $x_1 \mapsto 0$ (see Figure 4(b)). We tag the assignment $x_1 \mapsto 0$ as* implied*, since $x_1 \mapsto 1$ led to a conflict. Thus, we prevent that this assignment is reverted back to $x_1 \mapsto 1$ at a later point in time, which would lead to a non-terminating loop.*

When backtracking enough times, the search algorithm always yields a conflicting clause or a satisfying assignment and eventually exhausts all branches. However, always reverting the last decision made is not necessarily the best strategy, as the following example from [Har09] shows.

Example 3.4 *Consider the clauses $C_1 \equiv (\overline{x}_1\,\overline{x}_n\,x_{n+1})$ and $C_2 \equiv (\overline{x}_1\,\overline{x}_n\,\overline{x}_{n+1})$ as part of an unsatisfiable formula F. Exploring the trail $x_1\,x_2 \cdots x_{n-1}\,x_n$ leads to a conflict forcing us to backtrack and explore the trail $x_1\,x_2 \cdots x_{n-1}\overline{x}_n$. Since F is unsatisfiable, we are eventually (perhaps after further case-splits) forced to backtrack. Unfortunately, each time we change one of the assignments to $x_2, \ldots, x_{n-1}$, we will unnecessarily explore the case in which x_n is 1 again, since the solver is "unaware" of the fact that $x_1 \longrightarrow \overline{x}_n$ (which follows from* $\mathrm{Res}(C_1, C_2, x_{n+1})$*).*

The next section introduces *conflict clauses* as a means to prevent the repeated exploration of infeasible assignments.


x_1, guessed
$\overline{x}_4$, implied
$\overline{x}_2$, implied
x_3, implied
□
x_1, guessed
$\overline{x}_4$, implied
$\overline{x}_2$, implied
x_3, implied
□
$\overline{x}_1$, implied
$\overline{x}_2$, guessed
x_3, implied
backtrack
$x_1 \mapsto 0, x_2 \mapsto 0, x_3 \mapsto 1$

(a) Conflicting trail

(b) Trail after backtracking

Figure 4. Backtracking

3.5. Conflict-Driven Clause Learning

In their solvers GRASP and RELSAT, João Marques-Silva and Karen Sakallah [MSS96], and Roberto Bayardo and Robert Schrag [JS97], respectively, introduced a novel mechanism to analyse the conflicts encountered during the search for a satisfying assignment.

First, they partition trails into decision levels according to recursion depth of the case-splits performed.

Definition 3.4 (Decision Levels) *Each recursive application of the splitting rule gives rise to a new* decision level. *If a variable x is assigned 1 (owing to either a case split or unit propagation) at decision level n, we write $x@n$. Conversely, $\overline{x}@n$ denotes an assignment of 0 to x at decision level n.*

Secondly, the implications in a trail are represented using an *implication graph.*

Definition 3.5 (Implication Graph) *An implication graph is a labelled directed acyclic graph $G(V, E)$.*

- *The nodes V represent assignments to variables. Each $v \in V$ is labelled with a literal and its corresponding decision level.*
- *Each edge in an implication graph represents an implication deriving from a clause that is unit under the current partial assignment. Edges are labelled with the respective antecedent clauses of the assignment the edge points to.*
- *An implication graph may contain a single* conflict node *(indicated by the symbol □), whose incoming edges are labelled with the corresponding conflicting clause.*

Example 3.5 (Implication Graph for Example 3.2) *Figure 5 shows the implication graph for the trail presented in Example 3.2.*

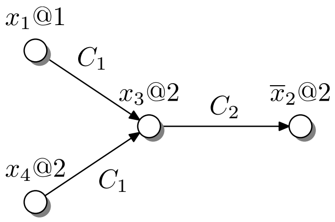


Figure 5. An implication graph for the trail in Table 3

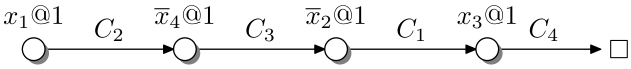


Figure 6. An implication graph with a conflict

If the implication graph contains a conflict, we can use it to determine the decisions that led to this conflict. Moreover, it enables us to derive a *conflict clause*, which, if added to the original formula, prevents the algorithm from repeating the decision(s) that led to the conflict.

Example 3.6 (Implication Graph with Conflict) *Figure 6 shows an implication graph for a trail emanating from the decision* $x_1 \mapsto 1$ *for the clauses*

$$C_1 \equiv (x_2\, x_3),\ C_2 \equiv (\overline{x}_1 \overline{x}_4),\ C_3 \equiv (\overline{x}_2 x_4),\ C_4 \equiv (\overline{x}_1 x_2 \overline{x}_3)\,.$$

The final node in the graph represents a conflict. The initial node of the graph is labelled with the decision that causes the conflict. Adding the unit clause $(\overline{x}_1)$ *to the original clauses guarantees that the decision* x_1 *will never be repeated.*

Example 3.7 *Figure 7 shows a partial implication graph for the clauses*

$$C_1 \equiv (\overline{x}_1 x_3 x_5), \quad C_2 \equiv (\overline{x}_1 x_2), \quad C_3 \equiv (\overline{x}_2 x_4), \quad \textit{and} \quad C_4 \equiv (\overline{x}_3 \overline{x}_4)$$

and the decisions $x_1@5$ *and* $\overline{x}_5@2$*. Using the implication graph, the decisions responsible for the conflict can be easily determined. Adding the conflict clause* $(\overline{x}_1 + x_5)$ *to the original formula rules out that this very combination of assignments is explored again.*

The advantage of *conflict clauses* over simple backtracking becomes clear when we revisit Example 3.4. Using an implication graph, we can quickly determine the assignments $x_1@1$ and $x_n@m$ which caused a conflict for either $C_1 \equiv (\overline{x}_1\, \overline{x}_n\, x_{n+1})$ or $C_2 \equiv (\overline{x}_1\, \overline{x}_n\, \overline{x}_{n+1})$. The conflict clause $(\overline{x}_1 + \overline{x}_n)$ eliminates this combination, pruning a large fraction of the search space which simple backtracking would have otherwise explored.

After adding a conflict clause, at least some of the decisions involved in the conflict need to be reverted (otherwise, the trail remains inconsistent with the clauses). Changing an assignment in the trail might invalidate all subsequently made decisions. Therefore, if we *backtrack* to a certain decision level n, we discard all decisions made at a level higher than n. It is clear that, of all decisions contributing to the conflict clause, we have to at least revert the one associated with the *current* decision level ($x_1@5$ in Example 3.7, for

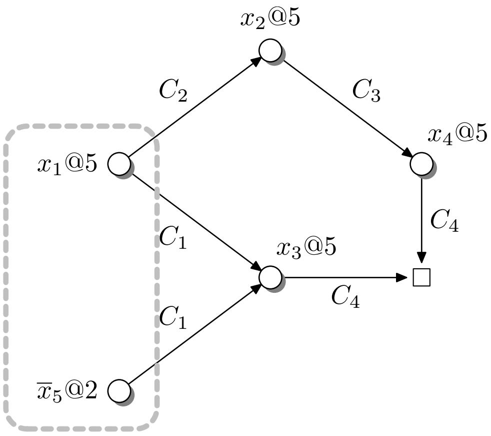


Figure 7. An implication graph for Example 3.7

instance). The *conflict-driven backtracking strategy* suggests to backtrack to the *second most recent decision level in the conflict clause* [MZM$^+$01] (level 2 in Example 3.7). This strategy has a compelling advantage: The conflict clause is *unit* (or *assertive*) under the resulting partial assignment. For instance, $(\overline{x}_1 + x_5)$ in Example 3.7 immediately implies $\overline{x}_1$ in this scenario.

3.6. Conflict Clauses and Resolution

Clause learning with conflict analysis does not impair the completeness of the search algorithm: even if the learnt clauses are dropped at a later point during the search, the trail guarantees that the solver never repeatedly enters a decision level with the same partial assignment.

We show the correctness of clause learning by demonstrating that each conflict clause is implied by the original formula. The following example is based on [KS08].

Example 3.8 (Conflict Clauses and Resolution) *Figure 8 shows a partial implication graph for the clauses*

$$C_1 \equiv (\overline{x}_4\, x_{10}\, x_6) \quad C_2 \equiv (\overline{x}_4\, x_2\, x_5) \quad C_3 \equiv (\overline{x}_5\, \overline{x}_6\, \overline{x}_7) \quad C_4 \equiv (\overline{x}_6\, x_7)\ .$$

The conflicting clause in this example is C_4. The immediate cause for the conflict are assignments x_6@6 and $\overline{x}_7$@6 to the literals $\overline{x}_6$ and x_7 of the clause C_4. These literals are implied by the clauses C_3 and C_1, respectively. Clearly, C_3 and C_4 (and C_1 and C_4) do not agree on the assignment of x_7 (and x_6, respectively). Accordingly, if we construct the resolvent *of C_3 and C_4 for the pivot x_7, we obtain a clause C_5:*

$$C_5 \quad \equiv \quad \mathrm{Res}(C_4, C_3, x_7) \quad \equiv \quad (\overline{x}_5\, \overline{x}_6)$$

While C_5 is certainly conflicting under the current partial assignment, we will not use it as a conflict clause: both x_5 and x_6 are assigned at decision level 6 and therefore C_5 is not assertive after backtracking.

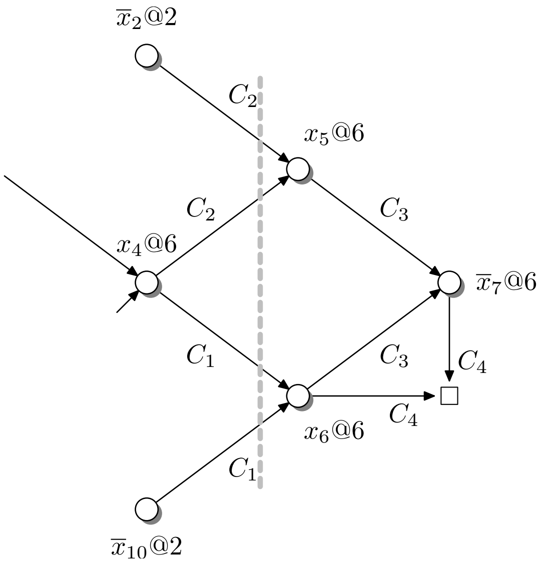


Figure 8. Conflict analysis and resolution

As previously mentioned, C_1 is the antecedent of x_6, and by a similar resolution step as before we obtain

$$C_6 \quad \equiv \quad \mathrm{Res}(C_5, C_1, x_6) \quad \equiv \quad (\overline{x}_4\,\overline{x}_5\,x_{10})\ .$$

Again, x_4 as well as x_5 are assigned at decision level 5. The clause C_2 is the antecedent of x_5, and we execute a final resolution step:

$$C_7 \quad \equiv \quad \mathrm{Res}(C_6, C_2, x_5) \quad \equiv \quad (x_2\,\overline{x}_4\,x_{10})$$

The resulting clause $(x_2\,\overline{x}_4\,x_{10})$ has the virtue of containing only one literal which is assigned at decision level 6 while still conflicting with the current partial assignment. Accordingly, if we backtrack to a decision level below 6, C_7 becomes assertive, forcing the solver to flip x_4. Therefore, we choose C_7 as conflict clause. Note that this clause corresponds to a cut (shown in Figure 8) that separates the (implied and guessed) decisions causing the conflict from the conflicting node.

We observe in Example 3.8 that it is possible to derive a conflict clause from the antecedents in the implication graph by means of resolution. These antecedents might in turn be conflict clauses. However, by induction, each conflict clause is implied by the original formula. Formal arguments establishing the completeness and correctness of clause learning and conflict analysis are provided in [MS95,MS99,Zha03].

The following example (based on the example presented in [MSS96]) demonstrates that, in general, there is a choice of assertive conflict clauses.

Example 3.9 *Consider the partial implication graph in Figure 9. Figure 10 shows three possible cuts that separate the decisions causing the conflict from the conflicting node. This results in three candidates for conflict clauses:*

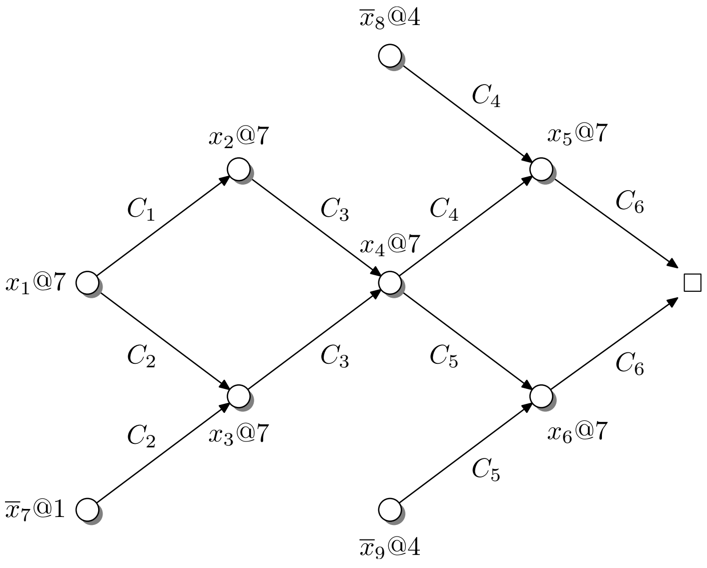


Figure 9. An implication graph with two *unique implication points*

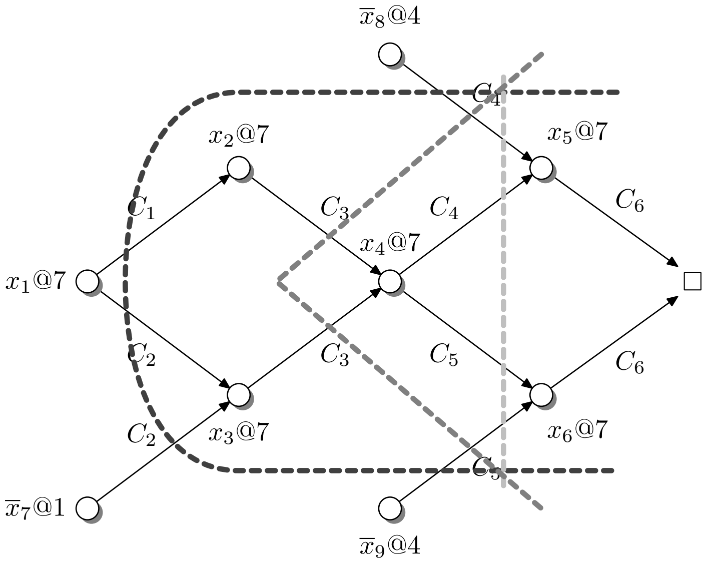


Figure 10. Possible cuts separating decision variables from the conflicting clause

1. $C_7 \equiv (x_8\,\overline{x}_1\,x_7\,x_9)$
2. $C_8 \equiv (x_8\,\overline{x}_4\,x_9)$
3. $C_9 \equiv (x_8\,\overline{x}_2\,\overline{x}_3\,x_9)$

We can dismiss the last clause, since it fails to be assertive after backtracking. The clauses $(x_8\,\overline{x}_1\,x_7\,x_9)$ *and* $(x_8\,\overline{x}_4\,x_9)$*, however, are viable candidates for a conflict clause.*

```
① If conflict at decision level 0 → UNSAT
② Repeat:
    ❶ if all variables assigned return SAT
    ❷ Make decision
    ❸ Propagate constraints
    ❹ No conflict? Go to ❶
    ❺ If decision level = 0 return UNSAT
    ❻ Analyse conflict
    ❼ Add conflict clause
    ❽ Backtrack and go to ❸
```

Figure 11. The DPLL algorithm with clause learning

The distinguishing property of clauses C_7 and C_8 when compared to clause C_9 in Example 3.9 is that the former two clauses contain only one literal assigned at the current decision level. This literal corresponds to a *unique implication point* (UIP).

Definition 3.6 (Unique Implication Point) *A unique implication point is any node (other than the conflict node) in the partial conflict graph which is on* all paths *from the decision node*[1] *to the conflict node of the current decision level.*

Accordingly, we can stop searching for a conflict clause (which is done by means of resolution) once we reach a unique implication point. But which UIP should we choose? We will base our choice on the following property of the conflict clause corresponding to the UIP *closest* to the conflict (referred to as the *first* UIP): by construction, the conflict clause induced by the first UIP *subsumes* any other conflict clause except for the asserting literal. For instance, in Example 3.9, $C_7 \equiv (x_8\,\overline{x}_1\,x_7\,x_9)$ contains all literals that occur in $C_8 \equiv (x_8\,\overline{x}_4\,x_9)$, except for the literal $\overline{x}_4$ which was assigned at decision level 6. Therefore, choosing C_8 as conflict clause has the following advantages:

1. The conflict clause C_8 is smaller than C_7, making it a more likely candidate for unit implications at a later point in the search algorithm.
2. Stopping at the first UIP has the lowest computational cost.
3. The second most recent decision level in the clause C_8 is at least as low as in any other conflict clause, which forces the solver to backtrack to a lower decision level.

The "first UIP" strategy is implemented in CHAFF [ZMMM01], whereas GRASP [MSS96], in contrast, learns clauses at all UIPs.

Figure 11 shows the complete DPLL algorithm with clause learning.

3.7. Decisions and Decision Heuristics

Step ②.❷ of the algorithm Figure 11 leaves the question of which variable to assign open. As we know from Section 3.3, this choice has no impact on the completeness of the search algorithm. It has, however, a significant impact on the performance of the solver, since this choice is instrumental in pruning the search space.

[1] The decision node of the current decision level is a unique implication point by definition.

3.7.1. 2 Literal Watching for Unit Propagation

The choice is clear as long as there are clauses that are unit under the current assignment. The book-keeping required to detect when a clause becomes unit can involve a high computational overhead if implemented naïvely, though. The authors of the CHAFF solver [MZM$^+$01] observed that it is sufficient to *watch* in each clause *any two* literals that have not been assigned, yet: a clause with m literals can only be unit (or conflicting) after at least $m - 1$ of its literals have been set to $\mathsf{0}$. Assignments to the non-watched literals can be safely ignored. When a variable is assigned $\mathsf{1}$, the solver only needs to visit clauses where the corresponding watched literal is negated. Each time one of the watched literals is assigned $\mathsf{0}$ the solver chooses one of the remaining unassigned literals to watch. If this is not possible, the clause is necessarily unit under the current partial assignment: any sequence of assignments that makes a clause unit will include an assignment of one of the watched literals. The computational overhead of this strategy is relatively low: in a formula with n clauses and m variables, $2 \cdot n$ literals need to be watched, and n/m clauses are visited per assignment on average. One of the key advantages of this approach is that the watched literals do not need to be updated upon backtracking. This is in contrast to the solver SATO [Zha97], for instance, which uses head and tail pointers that need to be updated whenever decisions are reverted.

In the case that no clauses are unit under the current partial assignment, however, it is necessary to choose a decision variable in step ②.❷ in Figure 11. In the following, we will discuss only a few such selection strategies; we refer the reader to [MS99] and [KS08] for a more complete overview over heuristics for choosing decision variables.

3.7.2. Dynamic Largest Individual Sum

It is conventional wisdom that it is advantageous to assign the most tightly constrained variables, i.e., variables that occur in a large number of clauses. On representative of such a selection strategy is known as the *dynamic largest individual sum* (DLIS) heuristic. At each decision point, it chooses the assignment that satisfies the most unsatisfied clauses. Formally, let p_x be the number of unresolved clauses containing x and n_x be the number of unresolved clauses containing $\overline{x}$. Moreover, let let x be variable for which p_x is maximal, and let y be variable for which n_y is maximal. If $p_x > n_y$, choose $\mathsf{1}$ as the value for x. Otherwise, choose $y \mapsto \mathsf{0}$. The disadvantage of this strategy is that the computational overhead is high: the algorithm needs to visit *all* clauses that contain a literal that has been set to true in order to update the values p_x and n_x for all variables contained in these clauses. Moreover, the process needs to be reversed upon backtracking.

3.7.3. Variable State Independent Decaying Sum

A heuristic commonly used in contemporary SAT solvers favours literals in recently added conflict clauses. Each literal is associated with a counter, which is initialised to zero. Whenever a (conflict) clause is added, its literals are *boosted*, i.e., the respective counters are increased. Periodically, all counters divided by constant, resulting in a decay causing a bias towards *recent* conflicts. At each decision point, the solver then chooses the unassigned literal with the highest counter (where ties are broken randomly by default). This approach, known as the *variable state independent decaying sum* (VSIDS) heuristics, was first implemented in the CHAFF solver [MZM$^+$01]. CHAFF maintains a list of unassigned literals sorted by counter. This list is only updated when conflict clauses are added, resulting in a very low overhead. Decisions can be made in constant time.

The emphasis on variables that are involved in recent conflicts leads to a *locality* based search, effectively focusing on sub-spaces [MZ09]. The sub-spaces induced by this decision strategy tend to coalesce, resulting in more opportunities for resolution of conflict clauses, since most of the variables are common.

Representing the counter using integer variables leads to a large number of ties. MINISAT avoids this problem by using a floating point number to represent the weight [ES04a]. Another possible (but significantly more complex) strategy is to concentrate *only* on unresolved conflicts by maintaining a stack of conflict clauses [GN02].

3.8. Unsatisfiable Cores

Given an unsatisfiable instance F, we can use the techniques described in Section 3.6 to construct a resolution refutation (see Definition 3.2 in Section 3.2). Intuitively, such a refutation identifies a *reason* for the inconsistency of the clauses in F. The clauses at the leaves of a resolution refutation are a subset of the clauses of F. By construction, the conjunction of these clauses is unsatisfiable.

Definition 3.7 (Unsatisfiable Core) *Given an unsatisfiable formula $F \equiv C_1 \cdot C_2 \cdots C_n$, any unsatisfiable subset of the set of clauses of F is an unsatisfiable core.*

Resolution proofs and unsatisfiable cores have applications in hardware verification [McM03]. Note that a formula typically does not have a unique unsatisfiable core. The following example demonstrates how we can use a SAT solver to construct an unsatisfiable core.

Example 3.10 (Constructing Unsatisfiable Cores) *Consider the following formula in conjunctive normal form:*

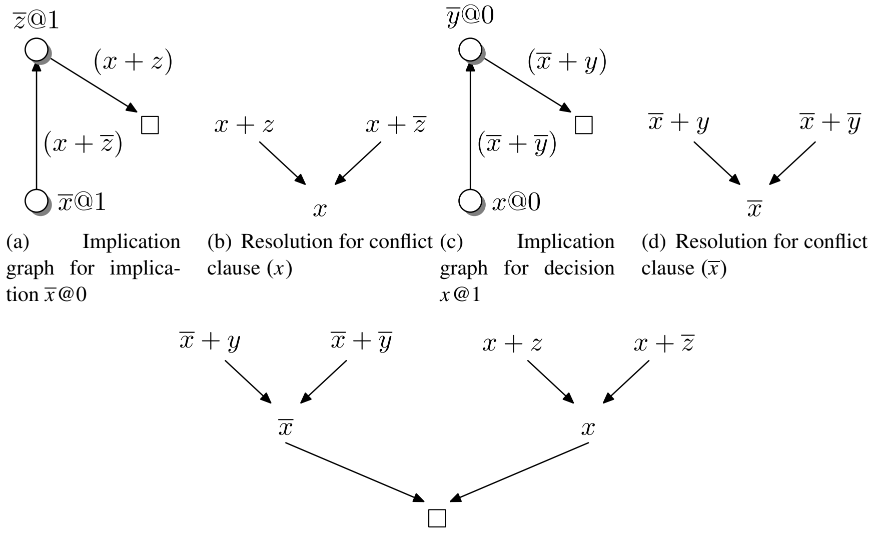
$$(\overline{x} + y) \cdot (\overline{x} + \overline{y}) \cdot (x + z) \cdot (x + \overline{z}) \cdot (z + y + \overline{x})$$

The problem instance does not contain unit literals, so the satisfiability solver is forced to make a decision. The VSIDS heuristic (see Section 3.7) assigns the highest priority to the literal $\overline{x}$. Accordingly, the solver assigns $x \mapsto 0$. This decision immediately yields a conflict, as depicted in Figure 12(a). Accordingly, the solver derives a conflict clause (x) – the justifying resolution step is shown in Figure 12(b). The conflict clause (x) forces the solver to assign $x \mapsto 1$ at decision level zero (x@0). Again, this leads to a conflict (see Figure 12(c)). The corresponding conflict clause is $(\overline{x})$ (see Figure 12(d)). This time, however, the conflict occurs at decision level zero and the satisfiability solver determines that the instance is unsatisfiable. The SAT solver finalises the resolution proof by resolving $(\overline{x})$ and (x) (see Figure 12(e)).

The unsatisfiable core

$$\{ \quad (\overline{x} + y),\ (\overline{x} + \overline{y}),\ (x + z),\ (x + \overline{z}) \quad \}$$

can be easily extracted from the resolution proof in Figure 12(e). The clause $(z + y + \overline{x})$ did not contribute to the contradiction and is therefore not contained in the core.



(a) Implication graph for implication $\overline{x}@0$

(b) Resolution for conflict clause (x)

(c) Implication graph for decision $x@1$

(d) Resolution for conflict clause $(\overline{x})$

(e) Final resolution proof

Figure 12. Construction of a resolution proof

An unsatisfiable core is minimal if removing any clause from the core makes the remaining set of clauses satisfiable.

Definition 3.8 (Minimal and Minimum Unsatisfiable Cores) *Let UC be an unsatisfiable core of the formula F (i.e., a set of clauses $UC \subseteq F$ such that $C_1 \cdot C_2 \cdots C_n \rightarrow 0$ if $C_i \in UC$ for $1 \leq i \leq n$). The unsatisfiable core UC is* minimal *if removing any one of its clauses C_i leaves the conjunction of the remaining clauses $UC \setminus C_i$ satisfiable. An unsatisfiable core is* minimum *if the original formula does not contain an unsatisfiable core UC_2 such that $|UC_2| < |UC|$.*

3.9. Incremental Satisfiability Solving

Many applications of SAT solvers require solving a sequence of similar instances which share a large number of clauses. Incremental satisfiability solvers [Str01,KSW01] support the reuse of learnt clauses in subsequent calls to the SAT solver when only a fraction of the clauses of the original problem have changed. To this end, an incremental solver drops all learnt clauses and reverts all decisions that derive from clauses that are part of the original instance but not of the subsequent related problem.

Example 3.11 *Recall the formula from Example 3.3:*

$$(x_2 + x_3) \cdot (\overline{x}_1 + \overline{x}_4) \cdot (\overline{x}_2 + x_4) \cdot (\overline{x}_1 + x_2 + \overline{x}_3)$$

Assume that the SAT solver derives the initial satisfying assignment

$$\{x_1 \mapsto 0, x_2 \mapsto 0, x_3 \mapsto 1, x_4 \mapsto 0\}$$

for this formula, which can be represented as the cube $\overline{x}_1 \cdot \overline{x}_2 \cdot x_3 \cdot \overline{x}_4$. *Note that at this point the SAT solver has learnt the clause* $(\overline{x}_1)$ *(c.f. Figure 4).*

Assume that in the next step we want to add the clause $(x_1 + x_2 + \overline{x}_3 + x_4)$ *(which happens to be the negation of* $\overline{x}_1 \cdot \overline{x}_2 \cdot x_3 \cdot \overline{x}_4$*) to the current set of clauses:*

$$(x_2 + x_3) \cdot (\overline{x}_1 + \overline{x}_4) \cdot (\overline{x}_2 + x_4) \cdot (\overline{x}_1 + x_2 + \overline{x}_3) \cdot \underbrace{(\overline{x}_1)}_{\textit{learnt}} \cdot \underbrace{(x_1 + x_2 + \overline{x}_3 + x_4)}_{\textit{new clause}}$$

Note that, while we have to revert the decisions made during the first run of the SAT solver, we are allowed to retain the learnt clause $(\overline{x}_1)$*, since it is a logical consequence of the original formula (i.e., at decision level 0). The SAT solver can now proceed to find a new satisfying assignment (e.g.,* $\{x_1 \mapsto 0, x_2 \mapsto 1, x_3 \mapsto 1, x_4 \mapsto 1\}$*). In this example, the new clause* $(x_1 + x_2 + \overline{x}_3 + x_4)$ *guarantees that this assignment differs from the previous one.*

3.10. Pre-processing Formulae

This section covers pre-processing techniques presented in [EB05] which enable us to reduce the size of the formula either before passing it to a satisfiability checker or during the search process.

3.10.1. Subsumption

A clause C_1 is said to subsume a clause C_2 if $C_1 \subseteq C_2$, i.e., all literals in C_1 also occur in C_2. If formula in CNF contains two clauses C_1 and C_2 such that C_1 subsumes C_2, then C_2 can be discarded. This is justified by the fact that, given a resolution proof, we can replace any occurrence of a clause C_2 by a clause C_1 which subsumes C_2 without invalidating the correctness of the proof. In fact, such a modification typically enables a reduction of the size of the proof [BIFH$^+$11].

3.10.2. Self-subsuming Resolution

Even though initial instance does not necessarily contain clauses subsuming others, such clauses may materialise during the search process. Eén and Biere [EB05] observes that formulae in CNF often contain clauses $(x + C_1)$ which *almost* subsume clauses $(\overline{x} + C_2)$ (where $C_1 \subseteq C_2$). After one resolution step we obtain the clause $\mathrm{Res}((x + C_1), (\overline{x} + C_2), x) = C_2$, which subsumes $(\overline{x}+C_2)$. Accordingly, the clause $(\overline{x}+C_2)$ can be dropped after resolution. Eén and Biere dubbed this simplification rule *self-subsuming resolution*.

Efficient data structures for implementing (self-)subsumption are presented in [EB05].

3.10.3. Variable Elimination by Substitution

Formulae that are encoded in CNF using the transformation introduced in Section 2.1.2 (or a similar approach) typically contain a large number of *functionally dependent* variables, namely the fresh variables introduced to represent terms (or gate outputs, respectively). In the following formula, for instance, the value of the variable x is completely determined by the values of y and z (c.f. Example 2.1):

$$\underbrace{(x \leftrightarrow (y \leftrightarrow z))}_{(\overline{x}+\overline{y}+z)\cdot(\overline{x}+\overline{z}+y)\cdot(\overline{y}+\overline{z}+x)\cdot(y+z+x)}$$

The algorithms previously presented are oblivious to this structural property and therefore fail to exploit it.Eén and Biere [EB05] presents an approach that eliminates dependent variables by substitution in an attempt to reduce the size of the resulting formula. First, note that the auxiliary variable x can be eliminated using the *rule for eliminating atomic formulae* introduced in Section 3.3. The application of this rule amounts to variable elimination by means of resolution. In general, given a set S of clauses all of which contain x, we can partition S into clauses containing x and clauses containing $\overline{x}$. Let $S_x \overset{\text{def}}{=} \{C \mid C \in S, x \in C\}$ and $S_{\overline{x}} \overset{\text{def}}{=} \{C \mid C \in S, \overline{x} \in C\}$. Abusing the notation we introduced in Section 3.2, we define

$$\text{Res}(S_x, S_{\overline{x}}, x) \overset{\text{def}}{=} \{\text{Res}(C_x, C_{\overline{x}}, x) \mid C_x \in S_x, C_{\overline{x}} \in S_{\overline{x}}\}\,.$$

A clause is trivial if it contains a literal and its negation. We observe that the pairwise resolution of the clauses corresponding to a definition of x introduced by the Tseitin transformation (see Table 2) yields *only* trivial clauses. We demonstrate this for the definition $x \leftrightarrow (y \leftrightarrow z)$ introduced in Example 2.1. Let

$$G \overset{\text{def}}{=} \{(\overline{x} + \overline{y} + z), (\overline{x} + \overline{z} + y), (\overline{y} + \overline{z} + x), (y + z + x)\}$$

denote the set of clauses introduced by the transformation. Splitting G as suggested above yields

$$G_{\overline{x}} = \{(\overline{x} + \overline{y} + z), (\overline{x} + \overline{z} + y)\} \qquad G_x = \{(\overline{y} + \overline{z} + x), (y + z + x)\}\,,$$

and we obtain

$$\text{Res}(G_x, G_{\overline{x}}, x) = \{(\overline{y} + z + \overline{z}), (\overline{y} + z + y), (\overline{z} + y + \overline{y}), (\overline{z} + y + z)\}\,.$$

The reader may verify that this holds for all transformations presented in Table 2. Accordingly, given a set of clauses S (all of which contain x) and the definition $G \subseteq S$ of x, we can safely omit the resolution steps $\text{Res}(G_x, G_{\overline{x}}, x)$. Let $R = S \setminus G$ be the *remaining* clauses that are not part of the definition of x. Then one can partition $\text{Res}(S_x, S_{\overline{x}}, x)$ into

$$\underbrace{\text{Res}(R_x, G_{\overline{x}}, x) \;\cdot\; \text{Res}(G_x, R_{\overline{x}}, x)}_{S''} \;\cdot\; \underbrace{\text{Res}(G_x, G_{\overline{x}}, x)}_{G'} \;\cdot\; \underbrace{\text{Res}(R_x, R_{\overline{x}}, x)}_{R'}\,.$$

In our example, G_x and $G_{\overline{x}}$ encode $x + \overline{(y \leftrightarrow z)}$ (i.e., $\overline{x} \rightarrow \overline{(y \leftrightarrow z)}$) and $\overline{x} + (y \leftrightarrow z)$, respectively. Accordingly, $\text{Res}(R_x, G_{\overline{x}}, x)$ can be interpreted as *substitution* of $(y \leftrightarrow z)$ for x in R_x (and similarly for $\text{Res}(G_x, R_{\overline{x}}, x)$). As a consequence, R' can be derived from S'' in a single hyper-resolution step (or a sequence of resolution steps) [GOMS04]. It is therefore admissible to replace S with S''.

Example 3.12 *Consider the CNF instance*

$$\underbrace{(x_1 + u)}_{R_{x_1}} \cdot \underbrace{(\overline{x}_1 + v)}_{R_{\overline{x}_1}} \cdot \underbrace{(\overline{x}_1 + \overline{y} + z) \;\cdot\; (\overline{x}_1 + \overline{z} + y)}_{G_{\overline{x}_1}} \cdot \underbrace{(\overline{y} + \overline{z} + x_1) \;\cdot\; (y + z + x_1)}_{G_{x_1}}\,.$$

We obtain

$$S'' \quad \equiv \quad (u+\overline{y}+z)\ \cdot\ (u+\overline{z}+y)\ \cdot\ (v+\overline{y}+\overline{z})\ \cdot\ (v+y+z)\,,$$

allowing us to reduce the size of the original formula by two clauses.

A more elaborate example for this approach is provided in Exercise 10 in Appendix A.

4. Boolean Satisfiability Checking: Extensions

After covering contemporary techniques to generate satisfying assignments or refutation proofs for propositional formulae in Section 3, we address a number of extensions of the SAT problem (Definition 3.1). As we will see, an in-depth understanding of the internals of SAT solvers is crucial to the techniques discussed in this section – the naïvely applying a SAT solver as a black box may result in a suboptimal performance of the resulting algorithm.

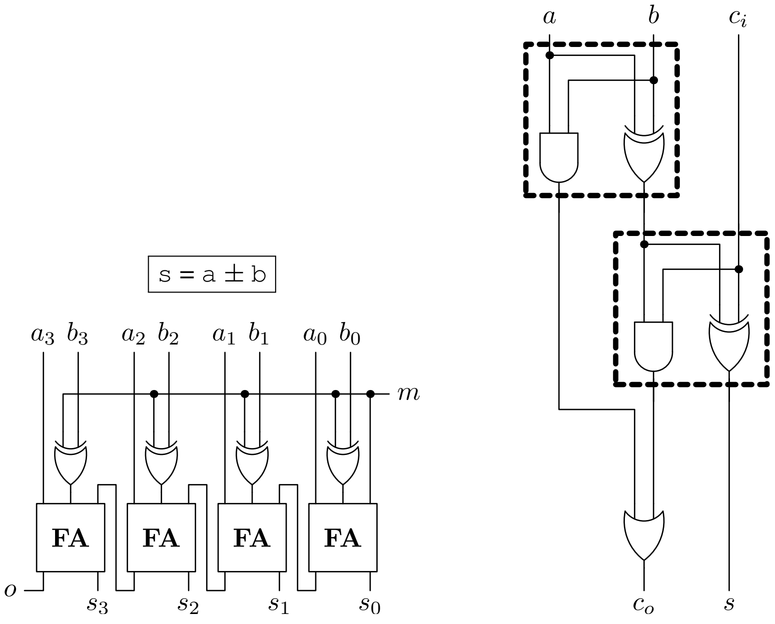
4.1. All-SAT

Given a satisfiable formula, the algorithms presented in Section 3 provide a single satisfying assignment. Some applications, however, require us to enumerate *all* satisfiable assignments of a formula [McM02]. It is easy to see that solving this problem is at least as hard as the Boolean satisfiability problem (Definition 3.1). In fact, the problem of determining the number of satisfying assignments of a formula is a prominent representative of the complexity class #P (see, for instance, [AB09]).

In practice, the problem can be tractable for certain instances, even though no polynomial algorithm is known. We can force the SAT solver to enumerate all satisfying assignments by subsequently *blocking* all assignments previously found. As explained in Section 2.1.1, any satisfying assignment $\mathcal{A}$ of a formula F can be represented as a *cube* over the variables of F. The negation of such a cube is a clause (by De Morgan's theorem). Adding this clause C to F effectively *blocks* the assignment, since C is clearly in conflict with the current assignment. C is therefore called a *blocking clause*. In fact, this approach has already been demonstrated in Example 3.11. To obtain all satisfying assignments, the process in Example 3.11 is repeated until the formula becomes unsatisfiable.

While we can take advantage of incremental satisfiability checking algorithms (c.f. Section 3.9), the size of the formula augmented with blocking clauses grows quickly. Moreover, blocking clauses which contain *all* variables of the original instance are less likely to become unit. Therefore, it is desirable to reduce the size of the blocking clause [McM02], i.e., to construct a smaller clause which still blocks the assignment $\mathcal{A}$. One possibility is to block the *decisions* that led to the current assignment (this information can be extracted from the trail described in Section 3.4). Let D be the cube representing these decisions. Clearly, $F \cdot D \leftrightarrow C_{\mathcal{A}}$, i.e., the decisions, in conjunction with the original formula, imply the single and unique assignment $\mathcal{A}$ (and *vice versa*). Moreover, $\overline{D} \rightarrow \overline{C_{\mathcal{A}}}$. Therefore $\overline{D}$ is a viable candidate for blocking $\mathcal{A}$.

An example application of All-SAT is presented in Example 4.7.



(a) Ripple carry adder

(b) Full adder

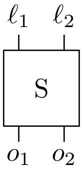
Figure 13. Encoding addition and subtraction in propositional logic

4.2. Cardinality Constraints

Satisfiability solvers are designed to work in the Boolean domain and do not support numeric reasoning *per se*. There is a number of applications for which it is desirable, however, to have at least rudimentary support for arithmetic over bounded domains. A common approach is to represent binary numbers using the two's complement system and to encode arithmetic operations using their corresponding circuit representation. Figure 13 shows the encoding of addition/subtraction as a ripple-carry-adder (Figure 13(a)), implemented as a chain of full adders (Figure 13(b)). This technique is known as eager bit-flattening. We refer the reader to [KS08] for a more detailed treatment of this topic.

Cardinality constraints are a common application of numerical constraints. Given a set $\{\ell_1, \ldots, \ell_n\}$ of literals, a cardinality constraint $((\sum_i \ell_i) \leq k)$ rules out all assignments in which more than k of these literals evaluate to 1 (here, $\sum$ denotes the arithmetic sum and not the logical "or" operator). This constraint can technically be encoded by constructing a circuit that computes $k - (\sum_i \ell_i)$ (using a tree of adder-subtractors depicted in Figure 13) and checking for arithmetic underflow. Such an encoding, however, introduces chains of exclusive-or gates. Note that exclusive-or is a non-monotonic operator (c.f. Table 1): a change of the value of a single variable occurring in a long chain of exclusive-or gates may propagate and necessitate an alteration of the values of a large number of subsequent variables in the chain (forced by unit-propagation), thus posing a challenge to contemporary satisfiability checkers.

Figure 14 shows a *sorting network* for two literals, an alternative way of encoding the constraint $((\sum_i \ell_i) \leq k)$ (where $i = 2$ in Figure 14). Intuitively, a sorting network shuffles all input values that are 1 "to the left", i.e., if m of the inputs of an n-bit sort-



ℓ_1	ℓ_2	o_1	o_2
0	0	0	0
0	1	1	0
1	0	1	0
1	1	1	1

$$o_1 \overset{\text{def}}{=} \ell_1 + \ell_2$$

$$o_2 \overset{\text{def}}{=} \ell_1 \cdot \ell_2$$

Figure 14. A sorting network for two literals

ing network (where $m \leq n$) are $\mathsf{1}$, then the output is a sequence of m ones followed by $n - m$ trailing zeroes. To encode an "at most k" constraint it is therefore sufficient to constrain the $(k+1)^{\text{th}}$ output signal to $\mathsf{0}$. The advantage of this construction over the previously discussed encoding is that sorting networks can be built entirely from (monotone) and-gates and or-gates (by cascading the circuit shown in Figure 14), thus avoiding the exclusive-or gates (and the associated snowball effect caused by unit-propagation, as described above) that retard the performance of the SAT solver. Sorting networks for n bits can be implemented using $O(n \cdot (\log n)^2)$ (see, for instance, [Par92]) or even $O(n \cdot \log n)$ gates [AKS83].

4.3. Maximum Satisfiability Problem (MAX-SAT)

Even if a formula F is unsatisfiable, there might still be assignments which satisfy a large number of its clauses. The *maximum satisfiability problem* (MAX-SAT) is concerned with finding the largest number of clauses that can be satisfied by some assignment.

Definition 4.1 (Maximum Satisfiability Problem) *Given a formula F in conjunctive normal form, determine the maximum number of clauses of F that can be satisfied by some assignment.*

If (and only if) the formula F is satisfiable, then there is an assignment that satisfies *all* of its clauses. Accordingly, the MAX-SAT problem is NP-hard. If, however, F is unsatisfiable, one needs to determine the largest subset of the clauses of F which, if conjoined, are still satisfiable. Equivalently, one can compute the smallest set of clauses that need to be dropped from the original instance to make it satisfiable.

Example 4.1 *Consider the unsatisfiable formula*

$$(\overline{r} + \overline{s} + t) \;\cdot\; (\overline{r} + s) \;\cdot\; (r) \;\cdot\; (\overline{t}) \;\cdot\; (s)\,. \tag{3}$$

Dropping the clause $(\overline{t})$ makes the instance satisfiable. Note that the largest set of satisfiable clauses is not unique: dropping the clause (r) also results in a satisfiable formula with four clauses as well.

The *partial* MAX-SAT problem is a generalisation of the MAX-SAT problem, in which some of the clauses are tagged as *hard* and must not be dropped.

Definition 4.2 (Partial Maximum Satisfiability Problem) *Given a formula F and a set $\{C_1, \ldots, C_m\} \subseteq F$ of* hard *clauses, determine the maximum number of clauses of F that can be satisfied by some assignment $\mathcal{A} \models C_1 \cdot C_2 \cdots C_m$.*

We refer to clauses of a partial MAX-SAT instance that are not hard as *soft* clauses.

Example 4.2 *We revisit Example 4.1, but require that the clauses* (r) *and* $(\overline{t})$ *of Formula (3) must not be dropped this time. In this scenario, dropping* $(\overline{r}+\overline{s}+t)$ *makes the formula satisfiable. Note that dropping either* $(\overline{r}+s)$ *or* (s) *does not yield a satisfiable instance.*

4.3.1. Relaxation Literals

The satisfiability checking techniques covered in Section 3 lack the ability to drop clauses. Contemporary satisfiability solvers such as MINISAT [ES04a], however, do at least provide the option to specify a partial assignment, which can be reverted in a subsequent call to the solver without sacrificing the learnt clauses that do not depend on this assignment.

As it turns out, this mechanism is sufficient to exclude clauses from the search process if we augment these clauses with so called *relaxation literals*. A relaxation literal is a literal over a variable v that does not occur in the original formula. If we replace a clause C_i that is part of the original formula with the *relaxed* clause $(v_i + C_i)$, the literal v_i acts as a switch which enables us to activate the clause C_i by setting v_i to 0. Conversely, the solver will ignore $(v_i + C_i)$ if v_i is set to 1 (by virtue of the *affirmative-negative rule* introduced in Section 3.3).

Example 4.3 *We continue working in the setting of Example 4.2. The following formula resembles Formula 3, except for the fact that the* soft *clauses have been augmented with the relaxation literals u, v, and w, respectively:*

$$(u+\overline{r}+\overline{s}+t) \cdot (v+\overline{r}+s) \cdot (r) \cdot (\overline{t}) \cdot (w+s)\,. \tag{4}$$

Now, any satisfiability solver can be used to determine that Formula 4 is satisfiable. The resulting satisfying assignment to u, v, and w determines which clauses were "dropped" by the solver.

Unfortunately, the technique outlined in Example 4.3 gives us no control over *which*, and more importantly, *how many* clauses the solver drops. Unless we modify the decision procedure, minimality is not guaranteed.

We can, however, restrict the number of dropped clauses by adding *cardinality constraints* (Section 4.2) to the relaxed formula. The corresponding constraint for the formula in Example 4.3, $(u + v + w) \leq 1$, instructs the SAT solver to drop *at most one* clause. Moreover, we already know that the solver has to drop *at least one* clause, since the original formula is unsatisfiable [MPLMS08]. In the case of Example 4.3, the SAT solver will find a satisfying solution. The rather restrictive cardinality constraint, however, does not account for (partial) MAX-SAT solutions that require the relaxation of more than one clause.

Example 4.4 *Consider the unsatisfiable formula*

$$(s) \cdot (\overline{s}) \cdot (t) \cdot (\overline{t})\,.$$

Note that this formula has two disjoint *unsatisfiable cores (c.f. Section 3.8). Accordingly, the formula*

$$(u+s) \cdot (v+\overline{s}) \cdot (w+t) \cdot (x+\overline{t}) \cdot (\sum\{u,v,w,x\} \leq 1)$$

is still *unsatisfiable.*

The formula in Example 4.4 requires the solver drop at least two clauses. This can be achieved by *replacing* the cardinality constraint with the slightly modified constraint $\sum\{u, v, w, x\} \leq 2$. As outlined in Section 4.2, this can be easily achieved by modifying a *single unit clause* as long as we use sorting networks to encode the constraint. Moreover, such a modification does not necessarily require us to restart the search from scratch, as mentioned in the first paragraph of Section 4.3.1. Incremental satisfiability solvers (see Section 3.9) are able to retain at least some of the clauses learnt from the first instance.

Accordingly, it is possible to successively relax the cardinality constraint in an efficient manner. If we follow this scheme, we obtain an algorithm to solve the partial MAX-SAT problem. If we successively increase the numeric parameter of the cardinality constraint (by forcing one single assignment of a literal of the sorting network), starting with one, we have a solution of the partial MAX-SAT problem readily at hand as soon as the SAT solver finds a satisfying assignment.

4.3.2. Core-Guided MAX-SAT

Example 4.5 *Consider the unsatisfiable formula*

$$(r+t) \cdot (r+s) \cdot (s) \cdot (\overline{s}) \cdot (t) \cdot (\overline{t}),$$

which resembles the formula in Example 4.4, except for the two clauses $(r + t)$ and $(r + s)$. Neither of these clauses influences the satisfiability of the formula. Accordingly, instrumenting these clauses with relaxation literals unnecessarily introduces additional variables and increases the size of the sorting network.

It is possible to avoid this unnecessary overhead in Example 4.5 by excluding the clauses $(r + t)$ and $(r + s)$ from the set of clauses the solver considers for removal (relaxation, respectively). However, how can we know that this is sound? The exclusion of a random clause may result in an invalid answer to the MAX-SAT problem.

The answer lies in the *minimal* unsatisfiable cores (Definition 3.8 in Section 3.8) of the formula. A clause C that is not contained in *any* (minimal) unsatisfiable core of F has no impact on the satisfiability of F. Accordingly, it is not necessary to instrument C with a relaxation literal. It is therefore possible to use cores to *guide* the selection of clauses to be relaxed [FM06] as demonstrated in the following example.

Example 4.6 *We continue working in the setting of Example 4.5. Following the method presented in Example 3.10, we obtain an initial core $\{(s), (\overline{s})\}$. Similar to Example 4.4, we instrument the clauses occurring this core with fresh relaxation literals and impose a cardinality constraint on these literals:*

$$(r+t) \cdot (r+s) \cdot (u+s) \cdot (v+\overline{s}) \cdot (t) \cdot (\overline{t}) \cdot (\sum\{u,v\} \leq 1) \qquad (5)$$

This relaxation "deactivates" the core (and also overlapping non-minimal cores, which demonstrates that the core guiding our instrumentation is not required to be minimal).

```
① While instance unsatisfiable, repeat:
    ❶ Obtain unsatisfiable core $UC$
    ❷ If $UC$ contains no soft clauses, return UNSAT
    ❸ For all soft clauses $\{C_1, \ldots, C_n\} \subseteq$ UC
        ∗ introduce fresh relaxation variable $v_i$
        ∗ $C_i = C_i \cup \{v_i\}$
    ❹ Add constraint $\left(\sum_{i=1}^{n} v_i\right) \leq 1$
② Obtain satisfying assignment $\mathcal{A}$
③ Return number of relaxation literals $v_i$ with $\mathcal{A}(v_i) = 1$
```

Figure 15. A core-guided MAX-SAT algorithm

The modified formula (5), however, is still not satisfiable. It contains a second core, namely $\{(t), (\overline{t})\}$. Defusing this core in a similar manner as the previous one yields

$$(r+t) \cdot (r+s) \cdot (u+s) \cdot (v+\overline{s}) \cdot (w+t) \cdot (x+\overline{t}) \cdot (\sum\{u, v\} \leq 1) \cdot (\sum\{w, x\} \leq 1)$$

A final run of the satisfiability solver yields a satisfying assignment which indicates that we need to relax two clauses. Note that it was not necessary to instrument the clauses $(r+t)$ and $(r+s)$ – this is a crucial advantage when it comes to large problem instances.

Figure 15 shows the pseudo-code of the core-guided MAX-SAT algorithm outlined in Example 4.6. Note that the introduction of relaxation literals complicates the use of incremental SAT algorithms (c.f. Section 3.9). At least the clauses learnt from hard constraints, however, can be retained across all instances.

4.4. Minimal Correction Sets (MCS)

In the previous section, the focus gradually shifted from clauses that can be satisfied simultaneously to clauses that need to be dropped to obtain a satisfiable formula. A set of clauses that has the latter property is also known as *minimal correction set* (MCS). The complement of each maxim*al* set of satisfiable clauses is an MCS. Accordingly, minimal correction sets are a generalisation of the MAX-SAT problem [LS09] – as the name indicates, we merely require minimality, i.e., in general, an MCS is not minim*um*.

Given this close relationship between the MAX-SAT problem and MCSes, it seems natural to extend the algorithm from Figure 15 to compute correction sets. Indeed, the algorithm readily provides one MCS (whose size, in fact, is minimum). But what if we desire to compute more than one, or even *all* MCSes? The technique presented in [LS09] is based on the algorithm in Section 4.3.2 and relies on blocking clauses (see Section 4.1) to exhaustively enumerate *all* minimal correction sets.

The algorithm in Figure 16 uses several auxiliary helper functions which implement techniques we have encountered in the previous sections.

- The procedure INSTRUMENT adds relaxation literals to clauses of the formula provided as parameter. If no second parameter is provided, the procedure instruments *all* clauses. Otherwise, the procedure only instruments clauses contained

```
① $k = 1$
② MCSes $= \emptyset$
③ $\text{UC}_k =$ unsatisfiable core of $F$
④ While (INSTRUMENT($F$) $\cdot$ (BLOCK(MCSes)) is satisfiable

    ❶ Instrument clauses in $\text{UC}_k$ with relaxation literal:
      $F_k = \text{INSTRUMENT}(F, \text{UC}_k) \cdot \text{ATMOST}(k, \text{UC}_k)$
    ❷ Enumerate satisfying assignments to relaxation variables:
      MCSes $=$ MCSes $\cup$ ALLSAT($F_k \cdot$ BLOCK($MCSes$))
    ❸ $\text{UC}_{k+1} = \text{UC}_k \cup$ core of $F_k \cdot$ BLOCK($MCSes$)
      (projected to clauses of $F$)
    ❹ $k = k + 1$

⑤ return MCSes
```

Figure 16. A core-guided algorithm to compute MCSes

in the set of clauses provided as second parameter. This process is outlined in Example 4.3.

- The procedure BLOCK adds blocking clauses that rule out the minimal correction sets provided as parameter. To this end, BLOCK adds one blocking clause for each MCS and assures thus that at least one clause of each MCS provided as parameter is *not* dropped.
- ATMOST generates a cardinality constraint which states that at most k clauses are dropped from the set of clauses provided as second parameter. (Cardinality constraints are discussed in Section 4.2.) Note that, unlike in the algorithm in Figure 15, which introduces one cardinality constraint per core, the algorithm in Figure 16 introduces only a single constraint. This improvement over [FM06] was first presented in [MSP08] and subsequently used in [LS09].
- ALLSAT enumerates all satisfying assignments to the relaxation literals contained in the formula provided as parameter. In our context, each of these assignments represents a minimal correction set. The respective techniques are covered in Section 4.1.

At the core of the algorithm in Figure 16 lies the MAX-SAT algorithm from Figure 15. In particular, the first intermediate result of the algorithm in Figure 16 is the set of all minim*um* correction sets, obtained by means of computing all solutions to the MAX-SAT problem. Subsequently, the algorithm gradually relaxes the cardinality constraint, allowing for correction sets of a larger cardinality while blocking MCSes found in previous iterations. In each iteration, the algorithm enumerates *all* correction sets of cardinality k. By induction, this guarantees the completeness of the algorithm; a formal argument is given in [LS09].

Example 4.7 *We recall the Formula 3 presented in Example 4.1:*

$$(\overline{r} + \overline{s} + t) \;\cdot\; (\overline{r} + s) \;\cdot\; (r) \;\cdot\; (\overline{t}) \;\cdot\; (s)$$

We simulate the algorithm in Figure 16 on this example. Since MCSes $= \emptyset$ in the initial iteration of the algorithm, the relaxed formula in line ④ is satisfiable. If we follow the *algorithm presented in Section 3.8, the satisfiability solver returns the unsatisfiable core* $\left\{(\overline{r}+\overline{s}+t)\,,(r),(\overline{t}),(s)\right\}$*. Accordingly, the algorithm constructs the formula*

$$(u+\overline{r}+\overline{s}+t)\;\cdot\;(\overline{r}+s)\;\cdot\;(v+r)\;\cdot\;(w+\overline{t})\;\cdot\;(x+s)\;\cdot(\textstyle\sum\{u,v,w,x\}\leq 1)$$

Then, it incrementally *constructs all satisfying assignments to* $\{u,v,w,x\}$ *that are consistent with this formula. We obtain the partial assignments*

$$\begin{aligned}&\{u\mapsto 1, v\mapsto 0, w\mapsto 0, x\mapsto 0\},\\&\{u\mapsto 0, v\mapsto 1, w\mapsto 0, x\mapsto 0\},\ \textit{and}\\&\{u\mapsto 0, v\mapsto 0, w\mapsto 1, x\mapsto 0\}\,.\end{aligned}$$

and the corresponding blocking clauses $(\overline{u})$*,* $(\overline{v})$*, and* $(\overline{w})$*. The respective MCSes of cardinality one are* $\{(\overline{r}+\overline{s}+t)\}$*,* $\{r\}$*, and* $\{\overline{t}\}$*. Note that the partial assignment* $\{u\mapsto 0, v\mapsto 0, w\mapsto 0, x\mapsto 1\}$ *is not a satisfying assignment, since dropping the clause* (s) *does not make the formula satisfiable – the unit clause* (s) *can be inferred from* $(\overline{r}+s)$ *and* (r)*. After blocking all MCSes (controlled by the variables* $\{u,v,w,x\}$*), we end up with the formula*

$$\begin{aligned}&(u+\overline{r}+\overline{s}+t)\;\cdot\;(\overline{r}+s)\;\cdot\;(v+r)\;\cdot\;(w+\overline{t})\;\cdot\;(x+s)\\&\qquad\cdot(\textstyle\sum\{u,v,w,x\}\leq 1)\;\cdot(\overline{u})\;\cdot\;(\overline{v})\;\cdot\;(\overline{w})\,.\end{aligned}$$

In step ❸*, the algorithm constructs the core of this formula, replaces the instrumented clauses with their original counterparts, and drops the cardinality constraint and the blocking clauses from the core. We obtain the new core*

$$\{(\overline{r}+\overline{s}+t)\,,(\overline{r}+s)\,,(r),(\overline{t})\}\,.$$

Note that, since the blocking clauses do not prevent (s) *from being dropped, the clause* $(\overline{r}+s)$ *must be contained in this core.*

In the next step, the algorithm increases k. Now, all clauses have to be instrumented (since the union of both cores computed so far happens to be the set of all clauses of the original formula), and all MCSes computed so far need to be blocked. In combination with the new cardinality constraint, we obtain

$$\begin{aligned}&(u+\overline{r}+\overline{s}+t)\;\cdot\;(y+\overline{r}+s)\;\cdot\;(v+r)\;\cdot\;(w+\overline{t})\;\cdot\;(x+s)\cdot\;(\overline{u})\;\cdot\;(\overline{v})\;\cdot\;(\overline{w})\\&\qquad\cdot(\textstyle\sum\{u,v,w,x,y\}\leq 2)\,.\end{aligned}$$

Since neither dropping (s) *nor dropping* $(\overline{r}+s)$ *from the original instance makes the formula satisfiable, the algorithm determines the satisfying assignment* $\{u\mapsto 0, y\mapsto 1, v\mapsto 0, w\mapsto 0, x\mapsto 1\}$*. This assignment is in fact the* only *satisfying partial assignment to the variables* $\{u,v,w,x,y\}$ *for the given formula. The corresponding blocking clause is* $(\overline{x}+\overline{y})$*.*

We leave it to the reader to verify that $\textsc{Instrument}(F)\cdot(\overline{u})\cdot(\overline{v})\cdot(\overline{w})\cdot(\overline{x}+\overline{y})$ *in line* ④ *is now unsatisfiable, and that the algorithm therefore terminates reporting the MCSes*

$$\{(\overline{r}+\overline{s}+t)\},\quad\{(r)\},\quad\{(\overline{t})\},\quad\textit{and}\quad\{(s),(\overline{r}+s)\}\,.$$

MCS	$(\overline{s})$	$(\overline{r}+s)$	(r)	(s)
$\{(\overline{s})\}$	$\boxed{\checkmark}$			
$\{(r),(s)\}$			$\boxed{\checkmark}$	$\boxed{\checkmark}$
$\{(s),(\overline{r}+s)\}$		$\boxed{\checkmark}$		$\boxed{\checkmark}$

Minimal unsatisfiable cores: $\boxed{\{(s),(\overline{s})\}}$ $\boxed{\{(r),(\overline{s}),(\overline{r}+s)\}}$

Figure 17. MCSes are hitting sets of minimal unsatisfiable cores, and vice versa

4.5. Minimal Unsatisfiable Cores

We observed in Section 4.4 that a minimal correction set comprises clauses that need to be dropped to "defuse" all unsatisfiable cores of a formula. Conversely, choosing at least one clause from each minimal correction set of a formula yields an unsatisfiable core. The following definition enables us to formalise this observation.

Definition 4.3 (Hitting Set) *Given a set of sets $\mathcal{S}$, a hitting set of $\mathcal{S}$ is a set H such that*

$$\forall S \in \mathcal{S} \,.\, H \cap S \neq \emptyset$$

Minimal correction sets and unsatisfiable cores are dual [LS08] in the following sense:

- Let $\mathcal{S}$ be the set of all MCSes of an unsatisfiable formula F. Then each (minimal) hitting set of $\mathcal{S}$ is a (minimal) unsatisfiable core (see Section 3.8).
- Let $\mathcal{S}$ be the set of all minimal unsatisfiable cores of an unsatisfiable formula F. Then each (minimal) hitting set of $\mathcal{S}$ is a (minimal) correction set for F.

The following example illustrates this duality.

Example 4.8 *The leftmost column in Figure 17 shows the set of all minimal correction sets $\{\{(\overline{s})\},\{(r),(s)\},\{(s),(\overline{r}+s)\}\}$ for the unsatisfiable formula*

$$F \quad \equiv \quad (\overline{s})\cdot(\overline{r}+s)\cdot(r)\cdot(s)\,.$$

The check-marks in the table indicate the occurrences of the clauses of F in the respective MCS. By choosing a subset of clauses of F which "hit" all *MCSes, we obtain a minimal unsatisfiable core. The formula F has two minimal unsatisfiable cores, namely $\{(s),(\overline{s})\}$ and $\{(r),(\overline{s}),(\overline{r}+s)\}$. The choice of appropriate "hitting" clauses is indicated in Figure 17 by oval and rectangular boxes, respectively.*

The problem of deciding whether a given set of sets has a hitting set of size k (or smaller) is NP-complete ([Kar72] in [LS08]). An algorithm optimised for the purpose of extracting cores from sets of MCSes can be found in [LS08].

Instead of presenting the algorithm suggested in [LS08], we draw the readers attention to the fact that after the final iteration of the algorithm in Figure 16, the set of clauses (BLOCK(MCSes)) in step ③ is a *symbolic* representation of all minimal correction sets. Essentially, we are looking for assignments that satisfy the CNF formula (Block(MCSes)). Note that the phase of all literals in (Block(MCSes)) is negative, since the respective clauses block assignments of 1 to relaxation variables. Accordingly, in order to find minimal unsatisfiable cores, we need to minimise the number of variables set to 0 in the satisfying assignment to (Block(MCSes)). Again, this can be achieved by means of gradually relaxed cardinality constraints.

Example 4.9 *In Example 4.7, we ended up with the blocking clause*

$$(\overline{u}) \cdot (\overline{v}) \cdot (\overline{w}) \cdot (\overline{x} + \overline{y}) \,,$$

where the relaxation literals u, v, w, x, and y correspond to the clauses $(\overline{r}+\overline{s}+t)$, (r), $(\overline{t})$, (s), and $(\overline{r} + s)$, respectively. Each of the clauses is satisfied if at least one of its literals evaluates to 1 *(and the corresponding variable evaluates to* 0*, respectively). In order to find a* minimal *hitting set, we constrain the literals using a cardinality constraint:*

$$(\overline{u}) \cdot (\overline{v}) \cdot (\overline{w}) \cdot (\overline{x} + \overline{y}) \cdot \left(\sum\{\overline{u}, \overline{v}, \overline{w}, \overline{x}, \overline{y}\} \leq k\right)$$

Note that k has to be at least four, since there are four clauses which do not share any literals. This threshold can be obtained using a syntactical analysis of the formula or simply by incrementally increasing k until it is sufficiently large.

If we generate all *minimal satisfying assignments to the constrained formula (using blocking clauses in a way similar to Example 4.7) we obtain the following assignments for $k = 4$:*

$$\{\overline{u} \mapsto 1, \overline{v} \mapsto 1, \overline{w} \mapsto 1, \overline{x} \mapsto 1, \overline{y} \mapsto 0\} \text{ and}$$
$$\{\overline{u} \mapsto 1, \overline{v} \mapsto 1, \overline{w} \mapsto 1, \overline{x} \mapsto 0, \overline{y} \mapsto 1\}$$

These assignments correspond to the minimal unsatisfiable cores

$$\{(\overline{r} + \overline{s} + t), (r), (\overline{t}), (s)\}$$
$$\{(\overline{r} + \overline{s} + t), (r), (\overline{t}), (\overline{r} + s)\}\,.$$

The hitting set problem is equivalent to the set cover problem, an NP-complete problem that has been extensively studied in complexity theory. We do not claim that the technique in Example 4.9 is competitive compared to other algorithms such as the one presented in [LS08] – the purpose of the example is to gain a deeper understanding of hitting sets.

The following section discusses examples of applications of the techniques presented in Sections 3 and 4.

5. Applications in Automated Verification

Contemporary SAT solvers are the enabling technology for a number of successful verification techniques. Bounded Model Checking, for instance, owes its existence to a large extent to the impressive advances of satisfiability solvers. This section presents – without claiming completeness in any way – a number of examples of how SAT solvers are applied in contemporary verification tools. After discussing how propositional logic can be used to represent circuits and software programs (Section 5.1), we discuss automated test-case generation (Section 5.2), Bounded Model Checking (Section 5.3), and fault localisation (Section 5.4).

5.1. Encoding Circuits and Programs

There is a natural correspondence between combinational circuits, such as the full-adder in Figure 13(b), and propositional logic formulae. Accordingly, the encoding of the circuit in Figure 13(b) is straight forward:

$$\begin{aligned}(o_1 \leftrightarrow (a \cdot b)) \;\cdot\; (o_2 \leftrightarrow (a \oplus b)) \;\cdot\; (o_3 \leftrightarrow (o_2 \cdot c_i)) \;\cdot \\ (s \leftrightarrow (o_2 \oplus c_i)) \;\cdot\; (c_o \leftrightarrow (o_1 + o_3))\end{aligned} \tag{6}$$

As described in the first step of Tseitin's encoding (presented in Section 2.1.2), the encoding introduces fresh variables o_1, o_2, o_3 that represent the inner signals and wires of the circuit that do not correspond to inputs or outputs. Based on Formula 6, we can construct a relation R which maps valuations to the input signals to the corresponding output signals:

$$R(\underbrace{a, b}_{\text{inputs}},\ \underbrace{s, c_o}_{\text{outputs}}) \overset{\text{def}}{=} \exists o_1\, o_2\, o_3\, . \begin{pmatrix} (o_1 \leftrightarrow (a \cdot b)) \cdot (o_2 \leftrightarrow (a \oplus b)) \cdot (o_3 \leftrightarrow (o_2 \cdot c_i)) \cdot \\ (s \leftrightarrow (o_2 \oplus c_i)) \cdot (c_o \leftrightarrow (o_1 + o_3)) \end{pmatrix} \tag{7}$$

Any satisfying assignment to this relation R (or to Formula 6, respectively) represents a possible combination of input/output signals, e.g.,

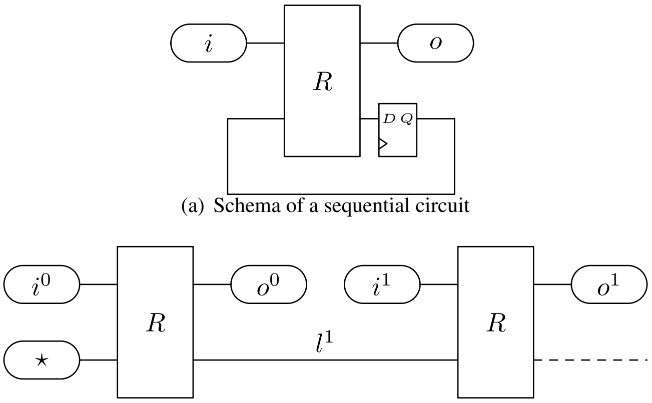
$$a \mapsto 0,\ b \mapsto 1,\ c_i \mapsto 1, s \mapsto 0, c_o \mapsto 1$$

corresponds to the case in which the full-adder yields a sum of zero and a set carry-out bit for an input of 0 and 1 and a carry-in bit of value 1. The *transition* relation R (7) is a symbolic encoding of *all* possible input/output pairs of the full-adder.

In a sequential circuit (see Figure 18(a)), the relation R encodes *one* execution cycle of the circuit. It is possible to extend this representation to a fixed number of k execution cycles by *replicating* (or *unfolding*) the combinational part of the circuit k times. The unfolding yields an *iterative logic array* [ABF90] (as illustrated in Figure 18(b) for two cycles). For each time-frame t, we introduce a fresh set of variables (as indicated by the super-script t). The initial state of the circuit imposes no constraints on the internal signals: their value can be either 1 or 0 (indicated by $\star$ in Figure 18(b)).

Figure 18 shows a simple example of such an unfolding. The sequential circuit in Figure 19(a) has two input signals i_1 and i_2 and one output signal o. In the corresponding 2-cycle unfolding in Figure 19(b) we introduce a fresh variable for each of these signals in each execution cycle (e.g., $i_1^1, i_1^2, \ldots$).

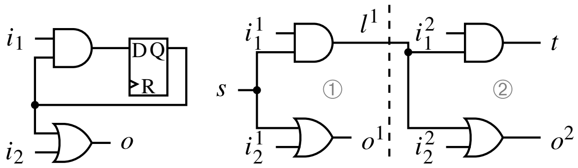
By means of Tseitin's transformation (Section 2.1.2) we obtain the propositional representation in Figure 20 of the unfolded circuit in Figure 19(b) in conjunctive normal



(a) Schema of a sequential circuit

(b) Unwinding of the circuit in Figure 18(a)

Figure 18. Unwinding circuits



(a) A sequential circuit

(b) A 2-cycle unfolding

Figure 19. A simple example of an unfolded circuit

cycle ①	$(\overline{l^1}\, i_1^1)\, (\overline{l^1}\, s)\, (\overline{i_1^1}\, \overline{s}\, l^1)$	$(\overline{i_2^1}\, o^1)\, (\overline{s}\, o^1)\, (\overline{o^1}\, i_2^1\, s)$
cycle ②	$(\overline{t}\, i_1^2)\, (\overline{t}\, l^1)\, (\overline{i_1^2}\, \overline{l^1}\, t)$	$(\overline{i_2^2}\, o^2)\, (\overline{l^1}\, o^2)\, (\overline{o^2}\, i_2^2\, l^1)$

Figure 20. Propositional encoding of the unwound circuit in Figure 19(b)

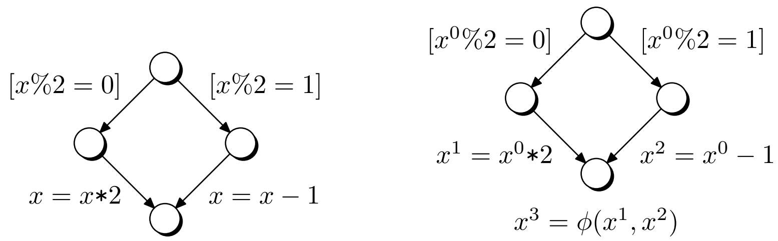
form. The clauses in Figure 20 are grouped with respect to the gates and cycles by which they are contributed.

Each satisfying assignment of this formula represents a feasible execution of two cycles of the sequential circuit in Figure 19(a). In general, k cycles are encoded by k instances of the relation R:

$$R(\vec{i}^1, \vec{o}^1) \cdot R(\vec{i}^2, \vec{o}^2) \cdots R(\vec{i}^k, \vec{o}^k) \tag{8}$$

(where $\vec{i}^t, \vec{o}^t$ represents the input and output variables of time-frame t).

Software programs can be encoded in a similar manner. The semantics of each instruction of a program is determined by the hardware implementation of the operators that occur in the instruction. The addition of two 4-bit variables a and b, for instance, can be encoded using the ripple-carry adder in Figure 13(a). Accordingly, each n-bit variable a in the program is encoded using n propositional variables representing the n bits of a;



(a) Conditional instruction (b) Static Single Assignment Form

Figure 21. Encoding the control flow of software programs

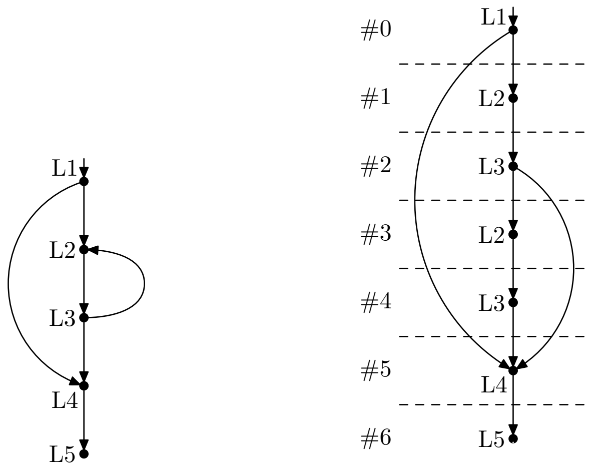
this technique is known as *bit-flattening* or *bit-blasting*. We refer the reader to [KS08] for a detailed treatment of various operators of common imperative programming languages such as ANSI-C or Java.

Accordingly, an instruction at the program location ℓ of the given program can be represented using a propositional relation $R_\ell(\vec{v}^i, \vec{v}^j)$, where $\vec{v}^i$ refers to the propositional variables representing the program state *before* the execution of the instruction and $\vec{v}^j$ refers to the variables representing the state *after* the execution. To avoid notational clutter, we refrain from using the bit-level representation of program instructions and will deploy a more abstract notation for transition relations (such as $R_\ell(x^i, x^j) \overset{\text{def}}{=} (x^j = x^i + 1)$ for the instruction x++ at location ℓ).

Unlike circuits, which are executed in a *synchronous* manner, software programs typically have a control flow structure which determines which instruction is executed at which point in time. Figure 21(a), for instance, illustrates the control flow graph (CFG) of the conditional instruction `if (x%2){x=x*2;}else{x--;}`. Accordingly, it is not sufficient to simply encode all instructions as propositional relations; one also has to take the control flow of the program into account.

In Figure 21(a), the variable x is assigned in two different branches of the conditional statement. We cannot simply use the same propositional variables to represent the value of x in each branch, since this would result in an unsatisfiable formula (x cannot take two values at the same time). Therefore, we need to guarantee, that different *versions* of the variables are used in each branch. This can be achieved by transforming the program into the *static single assignment* (SSA) form [CFR$^+$91]. The SSA form of a program is an intermediate representation used in compiler construction which guarantees that each variable is assigned exactly once. This property is achieved by replacing existing variables in the original program with fresh variables such that the right-hand side of each assignment in the program gets its own version of the assigned variable. The SSA form of the program fragment in Figure 21(a) is shown in Figure 21(b). In the SSA form, the assignment $x^3 = \phi(x^1, x^2)$ indicates a *join* of two branches of the program. At this point the variable x^3 needs to be assigned the value of either x^1 or x^2, depending on which branch was executed. It is, however, impossible to determine *statically* which branch modifies x. Therefore, we encode the control flow dependency into the transition relation as follows:

$$(x^1 = x^0*2) \cdot (x^2 = x^0 - 1) \cdot \Big((x^0\%2 = 0) \cdot (x^3 = x^1) + (x^0\%2 = 1) \cdot (x^3 = x^2)\Big)$$



(a) A CFG with a loop

(b) An unrolled CFG

Figure 22. Unwinding loops in software programs

```
while(x)        ⟶    if(x) {
  BODY;                 BODY;
                        if(x)
                          BODY;
                        else
                          exit();
                      }
```

Figure 23. An unrolling of a **while** loop. The exit statement terminates paths of depth greater than 2 .

In this formula, the value of x^3 depends on the value of x^0, the version of the variable x representing the value of x before the execution of the conditional statement.

Repetitive constructs (such as loops) can be treated similar to the encoding of execution cycles of a hardware circuit. However, instead of unwinding the entire transition relation of the program, each loop is unwound separately up to a pre-determined bound. Syntactically, this corresponds to a replication the loop body and the appropriate guard (Figure 23). The unwinding is illustrated in Figure 22. The CFG on the left side (Figure 22(a)) represents a simple program with a single loop. The graph on the right side (Figure 22(b)) illustrates the structure resulting from unwinding the loop body between the program locations `L3` and `L4` twice. The size of the resulting unwound program is linear in the size of the original program and the depth of the unwinding. Alternative unwinding techniques are discussed in the survey [DKW08]. After transforming the resulting unwound program into SSA, the bounded instance can be encoded as a propositional formula as before. The resulting formula effectively *simulates* all possible executions up to the pre-determined unwinding depth of the loops in the program. Accordingly, this technique is also known as *symbolic simulation* [Kin70].

5.2. Test-Case Generation

The fact that a bounded unwinding of a circuit design or a program symbolically encodes all possible executions up to a certain depth k makes it an ideal tool for automated test-case generation: each satisfying assignment generated by a SAT solver corresponds to a test scenario. In this setting, the circuit design or program takes the role of a specification; the resulting test-cases are used to verify the actual integrated circuit or a compiled version of the program. There is one subtle pitfall: a test-case extracted from the source code of the actual program (or chip design) under test must necessarily succeed if the compiler (or synthesis tool) is correct. It is therefore common practice to extract test-cases for the implementation from a *model* or abstract specification of the artifact under test. With the rise of model-based development (MBD) methodologies such models are available increasingly often. In [BHM$^+$10], for instance, test-cases are extracted from Simulink models. In combination with an incremental SAT solver it is possible to generate an entire suite of test-cases which satisfies certain coverage criteria such as path coverage or modified condition/decision coverage (MC/DC): the coverage criteria are simply encoded as constraints, and previously generated test-cases are barred by means of blocking clauses [HSTV08,HSTV09].

The test-case generator described in [BHM$^+$10], for instance, deploys *mutations* as a coverage criterion. A mutation is a small modification – such as using a wrong operator or variable name – to the original design or source code. A test suite which does not detect the injected fault is considered insufficient. Instead of using mutations to evaluate a given test-suite, however, [BHM$^+$10] uses mutations as a catalyst to generate a test-suite which, by construction, covers all injected faults. To this end, the test-generation algorithm contrasts the unwound transition relation of the *mutated* source code with the original unwound transition relation. Let R_m^k (R^k, respectively) denote the relation encoding k unwindings of the mutated (original) transition relation, respectively. The test-case generator then constructs the following formula:

$$R^k(\vec{i}^1, \vec{o}^k) \cdot R_m^k(\vec{i}^1, \vec{m}^k) \cdot \underbrace{\Big((o_1^k \oplus m_1^k) + (o_2^k \oplus m_2^k) + \cdots + (o_n^k \oplus m_n^k)\Big)}_{\text{miter}} . \qquad (9)$$

Observe that the *input* variables for both the original as well as the mutated transition relations are the same. A so called *miter* enforces that Formula 9 is only satisfiable if the valuation to the input variables yields output values $\vec{o}^k$ and $\vec{m}^k$ for the two different transition relations which disagree on at least one value. This approach (which is based on equivalence checking) guarantees that the resulting test case detects the faults injected in R_m^k.

Example 5.1 *Consider the mutated version (depicted in Figure 24) of the circuit in Figure 19(b). Note that the "and"-gate in the first cycle has been replaced with an "or"-gate, and that all output and internal signals were renamed. The input signals i_1^1, i_2^1, i_1^2, i_2^2, and the signal s representing the initial state of the latch remain unchanged.*

We obtain the following encoding in conjunctive normal form:

$$\begin{array}{lll} \textit{cycle} ① & (r^1\, \overline{i_1^1})\, (r^1\, \overline{s})\, (i_1^1\, s\, \overline{r^1}) & (\overline{i_2^1}\, m^1)\, (\overline{s}\, m^1)\, (\overline{m^1}\, i_2^1\, s) \\ \textit{cycle} ② & (\overline{u}\, i_1^2)\, (\overline{u}\, r^1)\, (\overline{i_1^2}\, \overline{r^1}\, t) & (\overline{i_2^2}\, m^2)\, (\overline{l^1}\, m^2)\, (\overline{m^2}\, i_2^2\, r^1) \end{array}$$

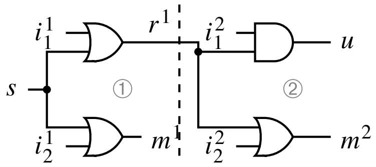


Figure 24. A mutated version of the circuit in Figure 19(b)

The miter $(o^1 \oplus m^1) + (o^2 \oplus m^2)$ *enforces that any assignment satisfying the conjunction of the formula representing the original circuit and the formula representing the mutated circuit yields different values for at least one output. The reader may verify that* o^2 *and* m^2 *must take different values in any extension of the partial assignment*

$$s \mapsto 0 \quad i_1^1 \mapsto 1 \quad i_2^1 \mapsto 0 \quad i_1^2 \mapsto 0 \quad i_2^2 \mapsto 0$$

to a total assignment. Accordingly, the corresponding test-case detects the incorrect "or"-gate.

5.3. Bounded Model Checking

Model checking [CGP99] is a technique that explores all reachable states of a model to check whether a given property is satisfied. Unlike testing, model checking performs an exhaustive search of the state space and provides a correctness guarantee that is rarely achieved by means of testing. Moreover, if the specification is violated, model checking tools are able to provide a *counterexample*, i.e., a witness demonstrating how the property in question can be violated.

Bounded model checking (BMC) [BCCZ99] is a variation of model checking which restricts the exploration to execution traces up to a certain (user-defined) length k. BMC either provides a guarantee that the first k execution steps of the model satisfy the property P or a counterexample of length at most k. This setting should sound familiar to the reader: Section 5.1 describes how all execution traces up to depth k can be encoded in a propositional formula. Given such an encoding, it is sufficient to augment each execution cycle with a formula encoding the negation of the property P. This is indicated in the following diagram (here, I denotes a constraint encoding the valid initial states of the model):

$$\begin{array}{ccccccccc} I & \cdot\ R & \cdot & R & \cdot & & \cdot & R & \\ & \longrightarrow & & \longrightarrow & & \cdots & & \longrightarrow & \\ \overline{P} & + & \overline{P} & + & \overline{P} & + & \overline{P} & + & \overline{P} \end{array}$$

Any assignment satisfying the resulting formula represents a counterexample to the claim that P holds. If the formula is unsatisfiable, on the other hand, then the claim holds in the first k execution steps. An example for a bounded model checking tool for programs written in the ANSI-C language is CBMC [CKL04].

5.4. Fault Localisation

The techniques discussed in Section 5.2 and Section 5.3 are aimed at discovering bugs in software and hardware designs. Finding a bug, however, is just the first step. Localising and understanding the underlying fault is often a much more daunting task.

Accordingly, automated support for fault localisation is highly desirable. In this section, we discuss how MAX-SAT and minimal correction sets can be applied to localise bugs. This approach, also known as *consistency-based diagnosis*, has been successfully applied to localise faults in hardware designs (see, for instance, [SMV$^+$07,SFBD08, FSBD08,CSVMS09,CSMSV10,ZWM11]) as well as in software [JM11].

Consistency-based diagnoses aims at identifying the fractions of the hardware design or the source code of the software that are *inconsistent* with an observed (or expected) behaviour. We distinguish two scenarios:

1. The transition relation R (obtained from the source code or hardware design) represents the *implementation* of the artifact under test and an observed behaviour of this implementation contradicts the specification (e.g., the requirements document or a use case scenario). This setting is addressed in [SMV$^+$07,SFBD08,FSBD08,CSVMS09,CSMSV10,JM11], for instance.
2. The transition relation R represents the *specification* of the artifact under test. The observed behaviour (e.g., a test run of a manufactured integrated circuit) is inconsistent with R. This scenario is addressed in [ZWM11].

In the first case we assume that the specification is given as a set of constraints or an assignment to the input and output variables of the transition relation R. In the second case we assume that the observed test scenario is provided as an assignment to the variables of R. While the two scenarios are in a sense dual, the objective in both cases is to identify the "elements" of the transition relation R that are inconsistent with the observed (or expected) behaviour. In both cases, we have to specify what we mean by "elements" – this is determined by the underlying *fault model*. The notion of a fault model is similar to the concept of a mutation, discussed in Section 5.2. A fault model determines which and how components of the transition relation can possibly fail. A gate in an integrated circuit, for instance, may malfunction and constantly yield an output value of 1 – this fault is known as *stuck-at-constant*. In a program, the developer may accidentally use the instruction `y=x++` instead of `y=++x`, which results in an incorrect value of the variable `y`. The faulty element is the respective gate in the first case and the respective line of code in the second case. In both cases, we would like to automatically pinpoint the location of the fault.

In the following, we consider only a rather simplistic fault model for hardware as well as for software: we assume that the output of a gate or the result of an assignment may be an *arbitrary* value. The motivation is that this fault model can be easily encoded in the transition relation R by means of *relaxation literals* (c.f. Section 4.3.1). By relaxing the set of clauses that encode the output value o_i of a gate or an assignment of an SSA variable x_i we effectively cut the signal or variable loose. The following two examples illustrate how minimal correction sets enable us to locate faults in this setting.

Example 5.2 *We work in scenario 1 described above. The code fragment in SSA in Figure 25 represents the implementation of a program. We assume that the specification of the software states that the value of y must be even after the execution of the conditional statement. This requirement is represented by the constraint* $(y^3\%2 = 0)$ *and obviously violated if* x^0 *is odd. Assume that the test engineer reports that the requirement does not hold for* $x^0 = 1$*. By combining the constraint and the assignment with the encoding of the program we obtain the formula*

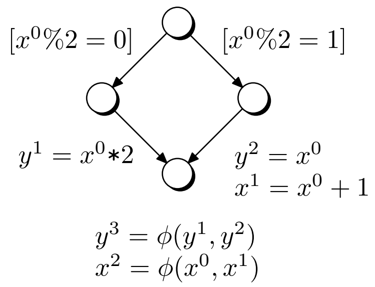


Figure 25. A conditional statement with a faulty branch

$$
\begin{aligned}
&(y^1 = x^0{*}2) \cdot (y^2 = x^0) \cdot (x^1 = x^0 + 1) \\
&\cdot \ \Big((x^0\%2 = 0) \cdot (x^2 = x^0) \cdot (y^3 = y^1) + (x^0\%2 = 1) \cdot (x^2 = x^1) \cdot (y^3 = y^2)\Big) \\
&\cdot (x^0 = 1) \cdot (y^3\%2 = 0) \ .
\end{aligned}
$$

Notably, this formula is unsatisfiable; for the given input the transition relation does not satisfy the requirement. In order to fix the bug, the developer needs to locate the fault. In accordance with the simplistic fault model suggested above, we assume that one of the assignments $y = x * 2$ or $y = x{+}{+}$ (represented by the expressions $(y^1 = x^0 * 2)$ and $(y^2 = x^0) \cdot (x^1 = x^0 + 1)$) are at fault. In order to locate the fault, we mark the propositional clauses encoding these expressions as soft clauses *and compute all minimal correction sets for the resulting formula. Note that, in accordance with our fault model, the conditions $(x^0\%2 = 0)$ and $(x^0\%2 = 1)$ are represented by hard clauses and may not be dropped. Moreover, the constraints $(x^0 = 1) \cdot (y^3\%2 = 0)$ representing the test scenario and the requirement must not be relaxed, either, since changing the test scenario or the requirements is an undesired solution to the problem.*

Using the algorithm described in Section 4.4, we can now compute the minimal correction sets for the problem instance described above. The $(y^2 = x^0)$ is identified as the culprit and helps the developer to narrow down the fault location to the instruction $y = x{+}{+}$.

Example 5.3 *In Scenario 2, the behaviour of the test artifact does not comply with the specification represented by R. This situation may arise in the context of* post-silicon validation, *for instance: the manufacturing process may introduce a fault in the prototype of a chip, resulting in a discrepancy of the behaviour of the integrated circuit and its design. Debugging an integrated circuit is non-trivial, since unlike in software debugging, its internal signals can not be easily observed.*

Consider the sequential circuit in Figure 19(a). After resetting the latch, we expect the output o to remain $\mathsf{0}$ as long as the input signal i_2 is constantly $\mathsf{0}$. Assume, however, that we observe an output value of $\mathsf{1}$ after two cycles when executing the described scenario on the chip. Figure 19(b) depicts a two-cycle unfolding of the circuit. Figure 20 shows the corresponding CNF encoding. Assume that we observe and record the values $o^1 \mapsto \mathsf{0}$ and $o^2 \mapsto \mathsf{1}$ during a test-run with the initial state $s \mapsto \mathsf{0}$ and the stimuli $i_2^1 \mapsto \mathsf{0}$, and $i_2^2 \mapsto \mathsf{0}$. Note that we have no information about the signal l^1. These observations contribute the hard constraint $\overline{o^1} \cdot o^2 \cdot \overline{s} \cdot \overline{i_2^1} \cdot \overline{i_2^2}$, which is not satisfiable in *conjunction with the formula in 20. Using a* MAX-SAT *solver, we can derive that the conjunction becomes satisfiable if we drop either* $(\overline{l^1}\, s)$ *or* $(\overline{o^2}\, i_2^2\, l^1)$ *(both of which are an MCS) from 20. Accordingly, either the "and"-gate in cycle one or the "or"-gate in cycle two must have defaulted. Notably, our fault localisation technique managed to narrow down the set of possibly faulty gates* without *knowledge about the internal signals of the physical circuit. Fault localisation in silicon debug is addressed in more detail in [ZWM11].*

6. Conclusion

The advances of contemporary SAT solvers have transformed the way we think about NP complete problems. They have shown that, while these problems are still unmanageable in the worst case, many instances can be successfully tackled. In Section 3, we discussed the main contributions and techniques that made this paradigm shift possible. Section 4 covers a number of natural extensions to the SAT problem, such as the enumeration all satisfying assignments (ALL-SAT) and determining the maximum number of clauses that can be satisfied by an assignment (MAX-SAT). SAT solvers and their extensions have immediate applications in domains such as automated verification, as discussed in Section 5. In fact, many successful verification techniques such as Bounded Model Checking owe their existence to the impressive advances of modern SAT solvers. While SAT solvers can easily be used as a black box, the realisation of many of these applications relies on internal features of SAT solvers and requires an in-dept understanding of the underlying algorithms.

SAT solvers are still improving at an impressive rate (as demonstrated by the results of the annual SAT solver competition – `http://www.satcompetition.org/`) and novel applications are conceived and published on a regular basis.

7. Acknowledgements

The authors are indebted to Matthias Schlaipfer and the attendees of the Summer School Marktoberdorf 2011 for their comments on this tutorial. This article has been published by IOS Press as a chapter [WM12] in [TAV12].

A. Exercises

Exercise 1 *Use Tseitin's transformation to convert $x + (y \cdot (\overline{z} \oplus x))$ into CNF.*

Solution By introducing the following fresh variables

$$\overbrace{x + \underbrace{(y \cdot \underbrace{(\overbrace{(\overline{z} \cdot \overline{x})}^{p} + \overbrace{(z \cdot x)}^{q})}_{u})}_{v}}^{w}$$

we obtain the formula

$$w \cdot (q \leftrightarrow (z \cdot x)) \cdot (p \leftrightarrow (\overline{z} \cdot \overline{x})) \cdot (u \leftrightarrow (p + q)) \cdot (v \leftrightarrow (y \cdot u)) \cdot (w \leftrightarrow (x + v))$$

We can now apply the rules

$$a \leftrightarrow (b + c) \equiv (\overline{b} + a) \cdot (\overline{c} + a) \cdot (\overline{a} + b + c) \tag{10}$$

$$a \leftrightarrow (b \cdot c) \equiv (\overline{a} + b) \cdot (\overline{a} + c) \cdot (\overline{b} + \overline{c} + a) \tag{11}$$

and get

$$\begin{aligned} w \cdot (\overline{q} + z) \cdot (\overline{q} + x) \cdot (\overline{z} + \overline{x} + q) \cdot (\overline{p} + \overline{z}) \cdot (\overline{p} + \overline{x}) \cdot (z + x + p) \cdot \\ (\overline{p} + u) \cdot (\overline{q} + u) \cdot (\overline{u} + p + q) \cdot (\overline{y} + v) \cdot (\overline{u} + v) \cdot (\overline{v} + y + u) \cdot \\ (\overline{x} + w) \cdot (\overline{v} + w) \cdot (\overline{w} + x + v) \end{aligned}$$

Exercise 2 *Follow the scheme in Table 2 in Section 2.1.2 to derive the Tseitin clauses that characterise the n-ary Boolean formulas $(y_1 + y_2 + \cdots + y_n)$ and $(y_1 \cdot y_2 \cdot \cdots \cdot y_n)$.*

Solution

- Disjunction:

$$\begin{aligned} & x \leftrightarrow (y_1 + y_2 + \cdots + y_n) \\ \equiv\; & (x \rightarrow (y_1 + y_2 + \cdots + y_n)) \cdot ((y_1 + y_2 + \cdots + y_n) \rightarrow x) \\ \equiv\; & (\overline{x} + y_1 + y_2 + \cdots + y_n) \cdot ((y_1 \rightarrow x) \cdot (y_2 \rightarrow x) \cdots (y_n \rightarrow x)) \\ \equiv\; & (\overline{x} + y_1 + y_2 + \cdots + y_n) \cdot (\overline{y}_1 + x) \cdot (\overline{y}_2 + x) \cdots (\overline{y}_n + x) \end{aligned}$$

- Conjunction:

$$\begin{aligned} & x \leftrightarrow (y_1 \cdot y_2 \cdot \cdots \cdot + y_n) \\ \equiv\; & (x \rightarrow (y_1 \cdot y_2 \cdot \cdots \cdot y_n)) \cdot ((y_1 \cdot y_2 \cdot \cdots \cdot y_n) \rightarrow x) \\ \equiv\; & ((\overline{x} + y_1) \cdot (\overline{x} + y_2) \cdot \cdots \cdot (\overline{x} + y_n)) \cdot \left(\overline{(y_1 \cdot y_2 \cdot \cdots \cdot y_n)} + x\right) \\ \equiv\; & ((\overline{x} + y_1) \cdot (\overline{x} + y_2) \cdot \cdots \cdot (\overline{x} + y_n)) \cdot \left(\overline{y}_1 + \overline{y}_2 + \cdots + \overline{y}_n + x\right) \end{aligned}$$

Exercise 3 *Which of the Boolean formulae below are satisfiable, and which ones are unsatisfiable?*

1. $x + x \cdot y$
2. $\overline{(x \cdot (x \rightarrow y)) \rightarrow y}$
3. $\overline{x \cdot ((x \rightarrow y) \rightarrow y)}$

Convert the formulae that are unsatisfiable into conjunctive normal form (either using Tseitin's transformation or the propositional calculus) and construct a resolution refutation proof.

Solution

- satisfiable: 1, 3
- unsatisfiable: 2

$$\begin{aligned}\overline{(x \cdot (x \rightarrow y)) \rightarrow y} &\equiv \overline{\overline{(x \cdot (\overline{x} + y))} + y} \\ &\equiv (x) \cdot (\overline{x} + y) \cdot (\overline{y})\end{aligned}$$

Resolution proof:

$$\mathrm{Res}((\overline{y}), \mathrm{Res}((x), (\overline{x} + y), x), y) \equiv \Box$$

Exercise 4 *Construct a resolution refutation graph for the following unsatisfiable formula:*

$$y_1 \cdot y_2 \cdot y_3 \ \cdot \ (\overline{y}_1 + x) \ \cdot \ (\overline{y}_2 + \overline{x} + z) \ \cdot \ (\overline{y}_3 + \overline{z})$$

Solution The resolution graph for Exercise 4 is shown in Figure 26.

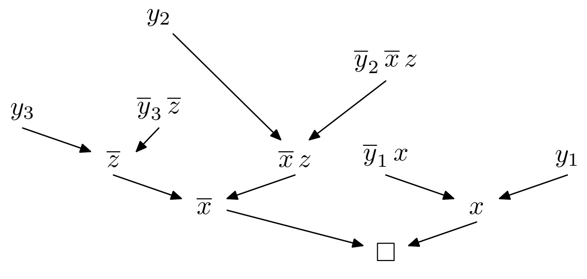


Figure 26. Resolution graph for Exercise 4

Exercise 5 *Apply the rules of the Davis-Putnam procedure (outlined in Section 3.3) to the following formula until you obtain an equi-satisfiable formula that cannot be reduced any further:*

$$y_1 \cdot y_2 \cdot (\overline{y}_1 + x + \overline{z}) \ \cdot \ (\overline{y}_2 + \overline{x} + z) \ \cdot \ (y_3 + \overline{z}) \ \cdot \ y_4$$

Solution We perform the following steps:

Step	Rule	Formula
1	*1-literal-rule* on y_1	$y_2 \cdot (x + \overline{z}) \cdot (\overline{y}_2 + \overline{x} + z) \cdot (y_3 + \overline{z}) \cdot y_4$
2	*1-literal-rule* on y_2	$(x + \overline{z}) \cdot (\overline{x} + z) \cdot (y_3 + \overline{z}) \cdot y_4$
3	*Affirmative-negative*	$(x + \overline{z}) \cdot (\overline{x} + z)$
4	*Resolution* on x	$(z + \overline{z})$

The resulting formula $(z + \overline{z})$ is a tautology and cannot be eliminated by any of the Davis-Putnam rules. Accordingly, the original formula must be satisfiable.

Exercise 6 *Apply the Davis-Putnam-Logeman-Loveland (DPLL) procedure (described in Section 3.4) to the following formula:*

$$y_1 \cdot y_2 \cdot (\overline{y}_1 + x + z) \cdot (\overline{y}_2 + \overline{x} + z) \cdot (y_3 + \overline{z}) \cdot (\overline{y}_3 + \overline{z})$$

Solution Table 4 shows one possible scenario. Note that there is no value of x that satisfies the formula. The reader may verify that choosing a decision variable other than x in the third step also yields a contradiction.

Partial Assignment	**Clauses**
$\{y_1 \mapsto 1\}$	$(y_2)\,(x\,z)\,(\overline{y}_2\,\overline{x}\,z)\,(y_3\,\overline{z})\,(\overline{y}_3\,\overline{z})$
$\{y_1 \mapsto 1, y_2 \mapsto 1\}$	$(x\,z)\,(\overline{x}\,z)\,(y_3\,\overline{z})\,(\overline{y}_3\,\overline{z})$
No more implications, we guess $x \mapsto 1$	
$\{y_1 \mapsto 1, y_2 \mapsto 1, x \mapsto 1\}$	$(z)\,(y_3\,\overline{z})\,(\overline{y}_3\,\overline{z})$
$\{y_1 \mapsto 1, y_2 \mapsto 1, x \mapsto 1, z \mapsto 1\}$	$(y_3)\,(\overline{y}_3)$
$\{y_1 \mapsto 1, y_2 \mapsto 1, x \mapsto 1, z \mapsto 1, y_3 \mapsto 1\}$	0
Contradiction, we have to revert $x \mapsto 1$	
$\{y_1 \mapsto 1, y_2 \mapsto 1, x \mapsto 0\}$	$(z)\,(y_3\,\overline{z})\,(\overline{y}_3\,\overline{z})$
$\{y_1 \mapsto 1, y_2 \mapsto 1, x \mapsto 0, z \mapsto 1\}$	$(y_3)\,(\overline{y}_3)$
$\{y_1 \mapsto 1, y_2 \mapsto 1, x \mapsto 0, z \mapsto 1, y_3 \mapsto 1\}$	0
Contradiction, no more decisions to undo	

Table 4. Assignment trail for Exercise 6

Exercise 7 *Simulate the* conflict-driven clause learning *algorithm presented in Section 3.5 on the following formula:*

$$\overbrace{(\overline{x} + \overline{y} + \overline{z})}^{C_0} \cdot \overbrace{(\overline{x} + \overline{y} + z)}^{C_1} \cdot \overbrace{(\overline{x} + y + \overline{z})}^{C_2} \cdot \overbrace{(\overline{x} + y + z)}^{C_3} \cdot$$

$$\overbrace{(x + \overline{y} + \overline{z})}^{C_4} \cdot \overbrace{(x + \overline{y} + z)}^{C_5} \cdot \overbrace{(x + y + \overline{z})}^{C_6} \cdot \overbrace{(x + y + z)}^{C_7}$$

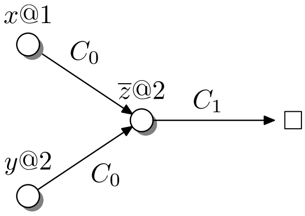


Figure 27. First implication graph arising in Exercise 7

Solution It is obvious that one has to make at least two decisions before one of the clauses becomes unit. If we start with the decisions $x@1$ and $y@2$, we obtain the implication graph in Figure 27.

By means of resolution (c.f. Section 3.6) we obtain the conflict clause $C_8 \equiv \text{Res}(C_0, C_1, z) \equiv (\overline{x} + \overline{y})$. We revert all decisions up to (but excluding) level 1, which is the second-highest decision level occurring in C_8. The clause C_8 is unit under the assignment $x@1$, thus implying the assignment $\overline{y}@1$. We obtain the implication graph in Figure 28. Again, there is a conflict.

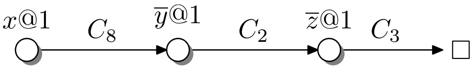


Figure 28. Second implication graph arising in Exercise 7

The resulting conflict clause is $C_9 \equiv \text{Res}(C_8, \text{Res}(C_2, C_3, z), y) \equiv (\overline{x})$, forcing us to revert to decision level zero and set x to 0. Under this assignment, none of the clauses is unit and we have to make a choice for either y or z. If we choose $y@1$, the clause C_4 becomes assertive and forces us to assign 0 to z. This assignment, however, is in conflict with C_5, and by means of resolution we obtain the conflict clause $C_{10} \equiv \text{Res}(C_4, C_5, z) \equiv (x + \overline{y})$.

C_{10} in combination with the unit clause C_9 yields $\overline{y}@0$. Under this assignment, the clause C_6 is unit, forcing us to assign 0 to z, which conflicts with clause C_7. Note that we obtained this conflict without making any decisions, i.e., we found a conflict at decision level zero. Accordingly, the formula is unsatisfiable.

Exercise 8 *Use the approach described in Section 3.6 to construct a resolution refutation proof for the formula presented in Exercise 7.*

Solution The solution to this exercise follows the steps of the solution to Exercise 7 and is left to the reader.

Exercise 9 *Find an unsatisfiable core of the formula*

$$(y) \cdot (x + \overline{y} + z) \cdot (\overline{x} + z) \cdot (\overline{x} + \overline{y}) \cdot (\overline{z} + \overline{y})\,.$$

(You are not allowed to provide the set of all clauses as a solution.)
Is your solution minimal?

Solution The set of clauses

$$\{(y), (x + \overline{y} + z), (\overline{x} + z), (\overline{z} + \overline{y})\}$$

forms a core of the formula in Exercise 9. This can be verified by means of resolution:

$$\begin{aligned}
&\mathrm{Res}((y), (x + \overline{y} + z), y) \equiv (x + z)\\
&\mathrm{Res}((x + z), (\overline{x} + z), x) \equiv (z)\\
&\mathrm{Res}((z), (\overline{z} + \overline{y}), z) =\equiv (\overline{y})\\
&\mathrm{Res}((\overline{y}), (y), y) = \square
\end{aligned}$$

Moreover, the core is minimal, since removing any one of the clauses "breaks" the core. Note that $\{(y), (x + \overline{y} + z), (\overline{x} + \overline{y}), (\overline{z} + \overline{y})\}$ is an alternative minimal solution.

Exercise 10 *Simplify the following formula using the substitution approach described in Section 3.10:*

$$\begin{aligned}
w \cdot (\overline{q} + z) \cdot (\overline{q} + x) \cdot (\overline{z} + \overline{x} + q) \cdot (\overline{p} + \overline{z}) \cdot (\overline{p} + \overline{x}) \cdot (z + x + p) \cdot\\
(\overline{p} + u) \cdot (\overline{q} + u) \cdot (\overline{u} + p + q) \cdot (\overline{y} + v) \cdot (\overline{u} + v) \cdot (\overline{v} + y + u) \cdot\\
(\overline{x} + w) \cdot (\overline{v} + w) \cdot (\overline{w} + x + v)
\end{aligned}$$

Solution Note that we do not know which clauses are "definitional" (i.e., introduce functionally dependent variables). In practice, this information is often not available and inferring it is computationally prohibitively expensive. Therefore we will not attempt to do so. Instead, we start by dividing the clauses into sets according to the positive and negative occurrences of the literals as shown in Figure 29.

Then, for each pair of sets S_ℓ, $S_{\overline{\ell}}$, we derive all possible resolvents and drop the resulting tautologies. If the resulting set of clauses $\mathrm{Res}(S_\ell, S_{\overline{\ell}}, \ell)$ is smaller than $S_\ell \cup S_{\overline{\ell}}$, we replace the clauses $S_\ell \cup S_{\overline{\ell}}$ with $\mathrm{Res}(S_\ell, S_{\overline{\ell}}, \ell)$. Otherwise, we retain the clauses $S_\ell \cup S_{\overline{\ell}}$. The set of resolvents of S_x and $S_{\overline{x}}$ has five elements:

$$\mathrm{Res}(S_x, S_{\overline{x}}, x) \equiv \{(\overline{q} + \overline{p}), (\overline{q} + w), (z + p + w), (\overline{w} + \overline{z} + v + q), (\overline{w} + \overline{p} + v)\}$$

This is one clause less than $S_x \cup S_{\overline{x}}$. Accordingly, replacing the clauses $S_x \cup S_{\overline{x}}$ with the corresponding set of resolvents reduces the size of the formula. This strategy is implemented in the SAT-solver MiniSAT [ES04b,EB05].

Exercise 11 *Use the core-guided algorithm presented in Section 4.3.2 to determine the solution of the partial MAX-SAT problem*

$$(\overline{x} + \overline{y}) \cdot (\overline{x} + z) \cdot (\overline{x} + \overline{z}) \cdot (\overline{y} + u) \cdot (\overline{y} + \overline{u}) \cdot (x) \cdot (y)\,,$$

where only the clauses (x) *and* (y) *may be dropped.*

$$
\begin{aligned}
S_x &= \{(\overline{q} + x), (z + x + p), (\overline{w} + x + v)\}\\
S_{\overline{x}} &= \{(\overline{z} + \overline{x} + q), (\overline{p} + \overline{x}), (\overline{x} + w)\}\\
S_y &= \{(\overline{v} + y + u)\}\\
S_{\overline{y}} &= \{(\overline{y} + v)\}\\
S_z &= \{(\overline{q} + z), (z + x + p)\}\\
S_{\overline{z}} &= \{(\overline{z} + \overline{x} + q), (\overline{p} + \overline{z})\}\\
S_p &= \{(z + x + p), (\overline{u} + p + q)\}\\
S_{\overline{p}} &= \{(\overline{p} + \overline{z}), (\overline{p} + \overline{x}), (\overline{p} + u)\}\\
S_q &= \{(\overline{z} + \overline{x} + q), (\overline{u} + p + q)\}\\
S_{\overline{q}} &= \{(\overline{q} + z), (\overline{q} + x), (\overline{q} + u)\}\\
S_u &= \{(\overline{p} + u), (\overline{v} + y + u), (\overline{q} + u)\}\\
S_{\overline{u}} &= \{(\overline{u} + p + q), (\overline{u} + v)\}\\
S_v &= \{(\overline{y} + v), (\overline{u} + v), (\overline{w} + x + v)\}\\
S_{\overline{v}} &= \{(\overline{v} + y + u), (\overline{v} + w)\}\\
S_w &= \{(w), (\overline{x} + w), (\overline{v} + w)\}\\
S_{\overline{w}} &= \{(\overline{w} + x + v)\}
\end{aligned}
$$

Figure 29. Positive and negative occurrences of literals

Solution Assume that the first unsatisfiable core we obtain is $\{(\overline{x} + \overline{y}), (x), (y)\}$. Accordingly, we augment the clauses (x) and (y) with relaxation variables and introduce a cardinality constraint which guarantees that at most one of these clauses is dropped:

$$(\overline{x} + \overline{y}) \cdot (\overline{x} + z) \cdot (\overline{x} + \overline{z}) \cdot (\overline{y} + u) \cdot (\overline{y} + \overline{u}) \cdot (r + x) \cdot (s + y) \cdot \sum(r, s) \leq 1$$

As illustrated in Figure 14, we can encode the constraint $\sum(r, s) \leq 1$ as $(\overline{r} + \overline{s})$, and we obtain the instance

$$(\overline{x} + \overline{y}) \cdot (\overline{x} + z) \cdot (\overline{x} + \overline{z}) \cdot (\overline{y} + u) \cdot (\overline{y} + \overline{u}) \cdot (r + x) \cdot (s + y) \cdot (\overline{r} + \overline{s}),$$

which is still unsatisfiable, since

$$
\begin{aligned}
&\mathrm{Res}((\overline{x} + z), (\overline{x} + \overline{z}), z) = (\overline{x}), && \mathrm{Res}((\overline{y} + u), (\overline{y} + \overline{u}), u) = (\overline{y}),\\
&\mathrm{Res}((r + x), (\overline{r} + \overline{s}), r) = (\overline{s} + x), && \mathrm{Res}((s + y), (\overline{s} + x), s) = (x + y),\\
&\mathrm{Res}((\overline{y}), (x + y), y) = (x), && \mathrm{Res}((\overline{x}), (x), x) = \square
\end{aligned}
$$

Accordingly, we add additional relaxation variables to the clauses $(r + x)$ and $(s + y)$ in the next iteration of the algorithm in Figure 15 and obtain

$$(\overline{x} + \overline{y}) \cdot (\overline{x} + z) \cdot (\overline{x} + \overline{z}) \cdot (\overline{y} + u) \cdot (\overline{y} + \overline{u}) \cdot (t + r + x) \cdot (v + s + y) \cdot \underbrace{(\overline{r} + \overline{s}) \cdot (\overline{t} + \overline{v})}_{\text{cardinality constraints}}$$

It is now possible for the satisfiability solver to relax both clauses (x) and (y) by choosing the assignment $\{t \mapsto \mathsf{1}, r \mapsto \mathsf{0}, v \mapsto \mathsf{0}, s \mapsto \mathsf{1}\}$, for instance. Accordingly, the algorithm in Figure 15 reports that two clauses need to be dropped to make the formula satisfiable.

Exercise 12 *Use the algorithm presented in Section 4.4 to derive all minimal correction sets for the unsatisfiable formula*

$$\overbrace{(x)}^{C_1} \cdot \overbrace{(\overline{x})}^{C_2} \cdot \overbrace{(\overline{x} + y)}^{C_3} \cdot \overbrace{(\overline{y})}^{C_4} \cdot \overbrace{(\overline{x} + z)}^{C_5} \cdot \overbrace{(\overline{z})}^{C_6} \; .$$

Solution (This example is presented in [LS08].) Due to the prioritisation of unit clauses, the first unsatisfiable core reported by the satisfiability checker is $\mathrm{UC}_1 \equiv \{(x), (\overline{x})\}$. By adding relaxation variables to all clauses of this core and by constraining the respective relaxation literals, we obtain the formula

$$(r_1 + x) \cdot (r_2 + \overline{x}) \cdot (\overline{x} + y) \cdot (\overline{y}) \cdot (\overline{x} + z) \cdot (\overline{z}) \cdot (\overline{r_1} + \overline{r_2})$$

Since dropping the clause $(\overline{x})$ does not yield a satisfiable instance, the ALLSAT procedure returns C_1 as the only MCS of size one. Accordingly, we block the corresponding assignment by adding the blocking clause $(\overline{r_1})$:

$$(r_1 + x) \cdot (r_2 + \overline{x}) \cdot (\overline{x} + y) \cdot (\overline{y}) \cdot (\overline{x} + z) \cdot (\overline{z}) \cdot (\overline{r_1} + \overline{r_2}) \cdot (\overline{r_1})$$

and obtain a new core $\{(\overline{r_1}), (r_1 + x), (\overline{x} + y), (\overline{y})\}$. Accordingly, $\mathrm{UC}_2 = \{C_1, C_2\} \cup \{C_1, C_3, C_4\}$, and we obtain the instrumented formula

$$(r_1 + x) \cdot (r_2 + \overline{x}) \cdot (r_3 + \overline{x} + y) \cdot (r_4 + \overline{y}) \cdot (\overline{x} + z) \cdot (\overline{z}) \cdot (\overline{r}_1) \cdot \sum(r_1, r_2, r_3, r_4) \leq 2$$

The ALLSAT algorithm determines all minimal correction sets for this formula. Note that the clause $(\overline{r}_1)$ prevents that the algorithm rediscovers the MCS $\{C_1\}$ in this step. Since $\mathrm{Res}((\overline{r}_{)}, (r_1 + x)) \equiv (x)$, blocking C_1 yields the formula

$$(x) \cdot (r_2 + \overline{x}) \cdot (r_3 + \overline{x} + y) \cdot (r_4 + \overline{y}) \cdot (\overline{x} + z) \cdot (\overline{z}) \cdot \sum(r_1, r_2, r_3, r_4) \leq 2\,,$$

which is unsatisfiable. We obtain the new core $\{C_1, C_5, C_6\}$ and execute the third iteration of the algorithm with $\mathrm{UC}_3 = \{C_1, C_2, C_3, C_4\} \cup \{C_1, C_5, C_6\}$. The corresponding instrumented and constrained version of the original formula is

$$(r_1 + x) \cdot (r_2 + \overline{x}) \cdot (r_3 + \overline{x} + y) \cdot (r_4 + \overline{y}) \cdot (r_5 + \overline{x} + z) \cdot (r_6 + \overline{z}) \cdot$$
$$\sum(r_1, r_2, r_3, r_4, r_5, r_6) \leq 3$$

In this iteration, we obtain the MCSes $\{C_2, C_3, C_5\}$, $\{C_2, C_3, C_6\}$, $\{C_2, C_4, C_5\}$, and $\{C_2, C_3, C_6\}$. Adding the corresponding blocking clauses to INSTRUMENT(F) results in an unsatisfiable instance and the algorithm terminates.

Exercise 13 *Derive all minimal unsatisfiable cores for the formula presented in Exercise 12.*

Solution The set of MCSes for the formula in Exercise 12 is

$$\{\{C_1\}, \{C_2, C_3, C_5\}, \{C_2, C_3, C_6\}, \{C_2, C_4, C_5\}, \{C_2, C_3, C_6\}\}\ .$$

We construct the corresponding minimal hitting sets as follows:

MCSes(F)	C_1	C_2	C_3	C_4	C_5	C_6
$\{C_1\}$	×					
$\{C_2, C_3, C_5\}$		×	×		×	
$\{C_2, C_3, C_6\}$		×	×			×
$\{C_2, C_4, C_5\}$		×		×	×	
$\{C_2, C_3, C_6\}$		×		×		×

Hitting sets: $\{C_1, C_2\}, \{C_1, C_3, C_4\}, \{C_1, C_5, C_6\}$

References

[AB09] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, 1st edition, 2009.

[ABF90] Miron Abramovici, Melvin A. Breuer, and Arthur D. Friedman. *Digital systems testing and testable design*. Computer Science Press, 1990.

[AKS83] M. Ajtai, J. Komlós, and E. Szemerédi. An $O(n \log n)$ sorting network. In *ACM Symposium on Theory of Computing (STOC)*, pages 1–9. ACM, 1983.

[BCCZ99] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer, 1999.

[BHM$^+$10] Angelo Brillout, Nannan He, Michele Mazzucchi, Daniel Kroening, Mitra Purandare, Philipp Rümmer, and Georg Weissenbacher. Mutation-based test case generation for Simulink models. In *Formal Methods for Components and Objects (FMCO) 2009*, volume 6286 of *Lecture Notes in Computer Science*, pages 208–227. Springer, 2010.

[BIFH$^+$11] Omer Bar-Ilan, Oded Fuhrmann, Shlomo Hoory, Ohad Shacham, and Ofer Strichman. Reducing the size of resolution proofs in linear time. *Software Tools for Technology Transfer (STTT)*, 13(3):263–272, 2011.

[Bus98] Samuel R. Buss. *Handbook of proof theory*. Studies in logic and the foundations of mathematics. Elsevier, 1998.

[CFR$^+$91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):451–490, 1991.

[CGP99] Edmund Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, December 1999.

[CKL04] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.

[Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. In *ACM Symposium on Theory of Computing (STOC)*, pages 151–158. ACM, 1971.

[CSMSV10] Yibin Chen, Sean Safarpour, Joao Marques-Silva, and Andreas Veneris. Automated design debugging with maximum satisfiability. *Transactions on CAD of Integrated Circuits and Systems*, 29:1804–1817, 2010.

[CSVMS09] Yibin Chen, Sean Safarpour, Andreas Veneris, and Joao Marques-Silva. Spatial and temporal design debug using partial MaxSAT. In *Great Lakes Symposium on VLSI*, pages 345–350. ACM, 2009.

[DKW08] Vijay D'Silva, Daniel Kroening, and Georg Weissenbacher. A survey of automated techniques for formal software verification. *Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 27(7):1165–1178, July 2008.

[DLL62] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5:394–397, July 1962.

[DP60] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7:201–214, July 1960.

[EB05] Niklas Eén and Armin Biere. Effective preprocessing in SAT through variable and clause elimination. In *Theory and Applications of Satisfiability Testing (SAT)*, volume 3569 of *Lecture Notes in Computer Science*, pages 102–104. Springer, 2005.

[ES04a] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In *Theory and Applications of Satisfiability Testing (SAT)*, volume 2919, pages 502–518. Springer, 2004.

[ES04b] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In *Theory and Applications of Satisfiability Testing (SAT)*, volume 2919 of *Lecture Notes in Computer Science*, pages 333–336. Springer, 2004.

[FM06] Zhaohui Fu and Sharad Malik. On solving the partial MAX-SAT problem. In *Theory and Applications of Satisfiability Testing (SAT)*, volume 4121 of *Lecture Notes in Computer Science*, pages 252–265. Springer, 2006.

[FSBD08] Görschwin Fey, Stefan Staber, Roderick Bloem, and Rolf Drechsler. Automatic fault localization for property checking. *Transactions on CAD of Integrated Circuits and Systems*, 27(6):1138–1149, 2008.

[GN02] E. Goldberg and Y. Novikov. Berkmin: A fast and robust SAT-solver. In *Design Automation and Test in Europe (DATE)*, pages 142–149. IEEE, 2002.

[GOMS04] Éric Grégoire, Richard Ostrowski, Bertrand Mazure, and Lakhdar Saïs. Automatic extraction of functional dependencies. In *Theory and Applications of Satisfiability Testing (SAT)*, volume 3542 of *Lecture Notes in Computer Science*. Springer, 2004.

[Har09] John Harrison. *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press, 2009.

[HSTV08] Andreas Holzer, Christian Schallhart, Michael Tautschnig, and Helmut Veith. FShell: Systematic test case generation for dynamic analysis and measurement. In *Computer Aided Verification (CAV)*, volume 5123 of *Lecture Notes in Computer Science*, pages 209–213. Springer, 2008.

[HSTV09] Andreas Holzer, Christian Schallhart, Michael Tautschnig, and Helmut Veith. Query-driven program testing. In *Verification, Model Checking and Abstract Interpretation (VMCAI)*, volume 5403 of *Lecture Notes in Computer Science*, pages 151–166. Springer, 2009.

[JM11] Manu Jose and Rupak Majumdar. Cause clue clauses: error localization using maximum satisfiability. In *Programming Language Design and Implementation (PLDI)*, pages 437–446. ACM, 2011.

[JS97] Roberto J. Bayardo Jr. and Robert Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Innovative Applications of Artificial Intelligence Conference (AAAI/IAAI*, pages 203–208, 1997.

[Kar72] Richard M. Karp. Reducibility among combinatorial problems. *Complexity of Computer Computations*, pages 85–103, 1972.

[Kin70] James C. King. *A program verifier*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1970.

[Kro67] M. R. Krom. The decision problem for a class of first-order formulas in which all disjunctions are binary. *Mathematical Logic Quarterly*, 13(1-2):15–20, 1967.

[KS08] Daniel Kroening and Ofer Strichman. *Decision procedures: An algorithmic point of view*. Texts in Theoretical Computer Science (EATCS). Springer, 2008.

[KSW01] Joonyoung Kim, Karem Sakallah, and Jesse Whittemore. SATIRE: A new incremental satisfiability engine. In *Design Automation Conference (DAC)*, pages 542–545. IEEE, 2001.

[LS08] Mark H. Liffiton and Karem A. Sakallah. Algorithms for computing minimal unsatisfiable subsets of constraints. *Journal of Automated Reasoning*, 40(1):1–33, 2008.

[LS09] Mark H. Liffiton and Karem A. Sakallah. Generalizing core-guided MAX-SAT. In *Theory and Applications of Satisfiability Testing (SAT)*, volume 5584 of *Lecture Notes in Computer Science*, pages 481–494. Springer, 2009.

[McM02] Kenneth L. McMillan. Applying SAT methods in unbounded symbolic model checking. In *Computer Aided Verification (CAV)*, volume 2404 of *Lecture Notes in Computer Science*, pages 250–264. Springer, 2002.

[McM03] Kenneth L. McMillan. Interpolation and SAT-based model checking. In *Computer Aided Verification (CAV)*, volume 2725 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 2003.

[MPLMS08] Paulo J. Matos, Jordi Planes, Florian Letombe, and João Marques-Silva. A MAX-SAT algorithm portfolio. In *European Conference on Artificial Intelligence*, volume 178 of *Frontiers in Artificial Intelligence and Applications*, pages 911–912. IOS Press, 2008.

[MS95] João Paulo Marques-Silva. *Search algorithms for satisfiability problems in combinational switching circuits*. PhD thesis, University of Michigan, 1995.

[MS99] João P. Marques-Silva. The impact of branching heuristics in propositional satisfiability algorithms. In *Progress in Artificial Intelligence, (EPIA)*, volume 1695 of *Lecture Notes in Computer Science*, pages 62–74. Springer, 1999.

[MSP08] João Marques-Silva and Jordi Planes. Algorithms for maximum satisfiability using unsatisfiable cores. In *Design Automation and Test in Europe (DATE)*, pages 408–413. IEEE, 2008.

[MSS96] João Paulo Marques-Silva and Karem A. Sakallah. GRASP – a new search algorithm for satisfiability. In *International Conference on Computer-aided Design (ICCAD)*, pages 220–227. IEEE, 1996.

[MZ09] Sharad Malik and Lintao Zhang. Boolean satisfiability from theoretical hardness to practical success. *Communications of the ACM*, 52(8):76–82, 2009.

[MZM$^+$01] Sharad Malik, Ying Zhao, Conor F. Madigan, Lintao Zhang, and Matthew W. Moskewicz. Chaff: Engineering an efficient SAT solver. *Design Automation Conference (DAC)*, pages 530–535, 2001.

[Par92] Ian Parberry. The pairwise sorting network. *Parallel Processing Letters*, 2:205–211, 1992.

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

[SFBD08] Andre Sülflow, Görschwin Fey, Roderick Bloem, and Rolf Drechsler. Using unsatisfiable cores to debug multiple design errors. In *Great Lakes Symposium on VLSI*, pages 77–82. ACM, 2008.

[Sha49] Claude E. Shannon. The synthesis of two-terminal switching circuits. *Bell Systems Technical Journal*, 28:59–98, 1949.

[SMV$^+$07] Sean Safarpour, Hratch Mangassarian, Andreas G. Veneris, Mark H. Liffiton, and Karem A. Sakallah. Improved design debugging using maximum satisfiability. In *Formal Methods in Computer-Aided Design (FMCAD)*, pages 13–19. IEEE, 2007.

[Str01] Ofer Strichman. Pruning techniques for the SAT-based bounded model checking problem. In *Correct Hardware Design and Verification Methods (CHARME)*, volume 2144 of *Lecture Notes in Computer Science*, pages 58–70. Springer, 2001.

[TAV12] *Tools for Analysis and Verification of Software Safety and Security*. NATO Science for Peace and Security Series. IOS Press, 2012.

[Tse83] G. Tseitin. On the complexity of proofs in poropositional logics. In J. Siekmann and G. Wrightson, editors, *Automation of Reasoning: Classical Papers in Computational Logic 1967–1970*, volume 2. Springer, 1983. Originally published 1970.

[WM12] Georg Weissenbacher and Sharad Malik. *Boolean Satisfiability Solvers: Techniques and Extensions*. In *NATO Science for Peace and Security Series* [TAV12], 2012.

[Zha97] Hantao Zhang. SATO: An efficient propositional prover. In *Conference on Automated Deduction (CADE)*, volume 1249 of *Lecture Notes in Computer Science*, pages 272–275. Springer, 1997.

[Zha03] Lintao Zhang. *Searching the Truth: Techniques for Satisfiability of Boolean Formulas*. PhD thesis, Princeton University, 2003.

[ZMMM01] Lintao Zhang, Conor F. Madigan, Matthew W. Moskewicz, and Sharad Malik. Efficient conflict driven learning in boolean satisfiability solver. In *International Conference on Computer-aided Design (ICCAD)*, pages 279–285, 2001.

[ZWM11] Charlie Shucheng Zhu, Georg Weissenbacher, and Sharad Malik. Post-silicon fault localisation using maximum satisfiability and backbones. In *Formal Methods in Computer-Aided Design (FMCAD)*. IEEE, 2011.