Formal Methods in Software Engineering

Boolean Satisfiability — Spring 2025

Konstantin Chukharev

§1 Foundations

Literals

Definition 1 (Literal): A *literal* is a propositional variable or its negation.

- p is a *positive literal*.
- $\neg p$ is a *negative literal*.

Definition 2 (Complement): The *complement* of a literal p is denoted by $\overline{p}$.

$$\overline{p} = \begin{cases} \neg p \text{ if } p \text{ is positive} \\ p \text{ if } p \text{ is negative} \end{cases}$$

Note: *complementary* literals p and $\overline{p}$ are each other's completement.

Clauses

Definition 3 (Clause): A *clause* is a disjunction of literals.

- An *empty clause* is a clause with no literals, commonly denoted by $\square$.
- A *unit clause* is a clause with a single literal.
- A *Horn clause* is a clause with at most one positive literal.

Note: $\square$ is *false in every interpretation*, that is, unsatisfiable.

Conjunctive Normal Form

Definition 4 (Conjunctive Normal Form (CNF)): A formula is said to be in *conjunctive normal form* if it is a conjunction of clauses.

$$A = \bigwedge_i \bigvee_j p_{ij}$$

Example: $A = (\neg p \vee q) \wedge (\neg p \vee q \vee r) \wedge p \wedge (\neg q \vee \neg r)$

Satisfiability on CNF

An interpretation ν satisfies a clause $C = p_1 \vee ... \vee p_n$ if it satisfies some (at least one) literal p_k in C.

An interpretation ν satisfies a CNF formula $A = C_1 \wedge ... \wedge C_n$ if it satisfies every clause C_i in A.

A CNF formula A is *satisfiable* if there exists an interpretation ν that satisfies A.

CNF Transformation

Any propositional formula can be converted to CNF by the repeated application of these rewriting rules:

1. $(A \to B) \Longrightarrow (\neg A \vee B)$
2. $(A \leftrightarrow B) \Longrightarrow (\neg A \vee B) \wedge (A \vee \neg B)$
3. $(A \oplus B) \Longrightarrow (A \vee B) \wedge (\neg A \vee \neg B)$
4. $\neg(A \wedge B) \Longrightarrow (\neg A \vee \neg B)$
5. $\neg(A \vee B) \Longrightarrow \neg A \wedge \neg B$
6. $\neg\neg A \Longrightarrow A$
7. $(A_1 \wedge ... \wedge A_n) \vee (B_1 \wedge ... \wedge B_m) \Longrightarrow (A_1 \vee B_1 \wedge ... \wedge B_m) \wedge ... \wedge (A_n \vee B_1 \wedge ... \wedge B_m)$

Theorem 1: If A' is obtained from a formula A by applying the CNF conversion rules, then $A' \equiv A$.

Problem with Exponential Blowup

Let's convert the following formula to CNF...

$$
\begin{aligned}
F &= p_1 \leftrightarrow (p_2 \leftrightarrow (p_3 \leftrightarrow (p_4 \leftrightarrow (p_5 \leftrightarrow p_6)))) \implies \\
&= (\neg p_1 \vee (p_2 \leftrightarrow (p_3 \leftrightarrow (p_4 \leftrightarrow (p_5 \leftrightarrow p_6))))) \wedge \\
&\quad (p_1 \vee \neg(p_2 \leftrightarrow (p_3 \leftrightarrow (p_4 \leftrightarrow (p_5 \leftrightarrow p_6))))) \implies \\
&= (\neg p_1 \vee (\neg p_2 \vee (p_3 \leftrightarrow (p_4 \leftrightarrow (p_5 \leftrightarrow p_6))))) \wedge \\
&\quad (\neg p_1 \vee (p_2 \vee \neg(p_3 \leftrightarrow (p_4 \leftrightarrow (p_5 \leftrightarrow p_6))))) \wedge \\
&\quad (p_1 \vee \neg(p_2 \leftrightarrow (p_3 \leftrightarrow (p_4 \leftrightarrow (p_5 \leftrightarrow p_6))))) \implies \ldots
\end{aligned}
$$

If we continue, the formula will *grow exponentially large*! The CNF of F consists of $2^5 = 32$ clauses.

There are formulas for which the minimum CNF has an exponential size.

Is there a way to avoid the exponential blowup? *Yes!*

Tseitin Transformation

A space-efficient way to convert a formula to CNF is the *Tseitin transformation*, which is based on so-called "*naming*" or "*definition introduction*", allowing to replace subformulas with the "*fresh*" (new) variables.

1. Take a subformula A of a formula F.
2. Introduce a new propositional variable n.
3. Add a *definition* for n, that is, a formula stating that n is equivalent to A.
4. Replace A with n in F.

Overall, construct $S := F[n/A] \wedge (n \leftrightarrow A)$

$$
\begin{aligned}
F &= p_1 \leftrightarrow (p_2 \leftrightarrow (p_3 \leftrightarrow (p_4 \leftrightarrow \overbrace{(p_5 \leftrightarrow p_6)}^{A}))) \implies \\
S &= p_1 \leftrightarrow (p_2 \leftrightarrow (p_3 \leftrightarrow (p_4 \leftrightarrow n))) \wedge \\
&\quad\ n \leftrightarrow (p_5 \leftrightarrow p_6)
\end{aligned}
$$

Note: The resulting formula is, in general, **not equivalent** to the original one, but it is *equisatisfiable*, i.e., it is *satisfiable* iff the original formula is satisfiable.

Equisatisfiability

Definition 5 (Equisatisfiability): Two formulas A and B are *equisatisfiable* if A is satisfiable *if and only if* B is satisfiable.

The set S of clauses obtained by the Tseitin transformation is *equisatisfiable* with the original formula F.

- Every model of S is a model of F.
- Every model of F can be extended to a model of S by assigning the values of fresh variables according to their definitions.

Avoiding the Exponential Blowup

Example: $F = p_1 \leftrightarrow (p_2 \leftrightarrow (p_3 \leftrightarrow (p_4 \leftrightarrow (p_5 \leftrightarrow p_6))))$

Applying the Tseitin transformation gives us:

$$\begin{aligned} S = {} & p_1 \leftrightarrow (p_2 \leftrightarrow n_3) \wedge \\ & n_3 \leftrightarrow (p_3 \leftrightarrow n_4) \wedge \\ & n_4 \leftrightarrow (p_4 \leftrightarrow n_5) \wedge \\ & n_5 \leftrightarrow (p_5 \leftrightarrow p_6) \end{aligned}$$

The equivalent CNF of F consists of $2^5 = 32$ clauses.

The equisatisfiable CNF of F consists of 4 clauses, yet introduces 3 fresh variables.

Clausal Form

Definition 6 (Clausal Form): A *clausal form* of a formula F is a set S_F of clauses which is satisfiable iff F is satisfiable.

A clausal form of a *set* of formulas F' is a set S' of clauses which is satisfiable iff F' is satisfiable.

Even stronger requirement:

- F and S_F have the same models in the language of F.
- F' and S' have the same models in the language of F'.

The main advantage of the clausal form over the CNF is that we can convert any formula into a set of clauses in *almost linear time*.

1. If F is a formula which has the form $C_1 \wedge ... \wedge C_n$, where $n > 0$ and each C_i is a clause, then its clausal form is $S \stackrel{\text{def}}{=} \{C_1, ..., C_n\}$.
2. Otherwise, apply Tseitin transformation: introduce a name for each subformula A of F such that B is not a literal and use this name instead of a subformula.

§2 SAT

Boolean Satisfiability Problem (SAT)

SAT is the classical NP-complete problem of determining whether a given Boolean formula is *satisfiable*, that is, whether there exists an assignment of truth values to its variables that makes the formula true.

$$\exists X.f(X) = 1$$

SAT is a *decision* problem, which means that the answer is either "yes" or "no". However, in practice, we are mainly interested in *finding* the actual satisfying assignment if it exists — this is a *functional* SAT problem.

Historically, SAT was the first problem proven to be NP-complete, independently by Stephen Cook [1] and Leonid Levin [2] in 1971.

Theorem 2 (Cook–Levin): SAT is NP-complete.

That is, any problem in NP can be *reduced* to SAT in polynomial time, using many-one reductions.

Solving General Search Problems with SAT

Modelling and solving general search problems:

1. Define a finite set of possible *states*.
2. Describe states using propositional *variables*.
3. Describe *legal* and *illegal* states using propositional *formulas*.
4. Construct a propositional *formula* describing the desired state.
5. Translate the formula into an *equisatisfiable* CNF formula.
6. If the formula is *satisfiable*, the satisfying assignment corresponds to the desired state.
7. If the formula is *unsatisfiable*, the desired state does not exist.

Example: Graph Coloring

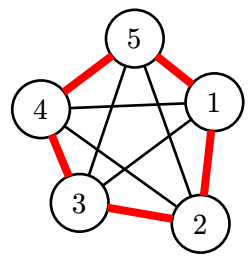
Recall that a graph $G = (V, E)$ consists of a set V of vertices and a set E of edges, where each edge is an unordered pair of vertices.

A complete graph on n vertices, denoted K_n, is a graph with $|V| = n$ such that E contains all possible pairs of vertices. In total, K_n has $\frac{n(n-1)}{2}$ edges.

Given a graph, color its vertices such that no two adjacent vertices have the same color.

Given a complete graph K_n, color its edges using k colors without creating a monochromatic triangle. What is the largest complete graph for which this is possible for a given number of colors?

- For $k = 1$, the answer is $n = 2$.
  - The graph K_2 has only one edge, which can be colored with a single color.
- For $k = 2$, the answer is $n = 5$.
- For $k = 3$, the answer is $n = 16$.



Modelling and Solving the Graph Coloring Example

1. *Define a finite set of possible states.*
   - Each possible edge coloring is a state. There are $3^{|E|}$ possible states.
2. *Describe states using propositional variables.*
   - A simple (*one-hot*, or *direct*) encoding uses three variables for each edge: e_1, e_2, and e_3. There are 9 possible combinations of values of three variables, which given a state space of $9^{|E|}$. This is larger than necessary, but keeps the encoding simple.
3. *Describe legal and illegal states using propositional formulas.*
   - For each edge $e \in E$, the formula $e_1 + e_2 + e_3 = 1$ (so called "cardinality constraint") ensures that each edge is colored with exactly one color. This reduces the state space to $3^{|E|}$.
4. *Construct a propositional formula describing the desired state.*
   - The desired state is one in which there are no monochromatic triangles. For each triangle (e, f, g), we explicitly forbid it from being colored with the same color:

$$\neg((e_1 \leftrightarrow f_1) \wedge (f_1 \leftrightarrow g_1) \wedge (e_2 \leftrightarrow f_2) \wedge (f_2 \leftrightarrow g_2) \wedge (e_3 \leftrightarrow f_3) \wedge (f_3 \leftrightarrow g_3))$$

Modelling and Solving the Graph Coloring Example [2]

5. *Translate the formula into an equisatisfiable CNF formula.*
   - This can be done using the Tseitin transformations.
6. *If the formula is satisfiable, the satisfying assignment corresponds to the desired state.*
   - The satisfying assignment corresponds to a valid edge coloring. Among variables e_1, e_2, and e_3, the single one with the value of 1 corresponds to the color of the edge.
7. *If the formula is unsatisfiable, the desired state does not exist.*
   - If the formula is unsatisfiable, there is no valid edge coloring.

Now, run a SAT solver for increasing values of n, and find the largest n for which the formula is satisfiable. The answer is $n = 16$ for $k = 3$.

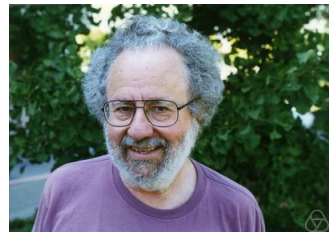
§3 Algorithms for SAT

Davis–Putnam Algorithm

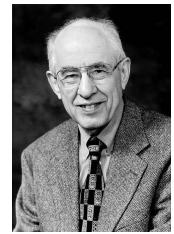
The first algorithm for solving the SAT problem was proposed by Martin Davis and Hilary Putnam in 1960 [3].

Satisfiability-preserving transformations:

- The 1-literal rule (**unit propagation**).
- The affirmative-negative rule (**pure literal**).
- The atomic formula elimination rule (**resolution**).



Martin Davis



Hilary Putnam

The first two rules reduce the total number of literals in the formula. The third rule reduces the number of variables in the formula. By repeatedly applying these rules, we can simplify the formula until it becomes *trivially* satisfiable (formula without clauses) or unsatisfiable (formula containing an empty clause).

Hereinafter, we assume that the formulas are given in CNF form.

Unit Propagation Rule

Definition 7 (Unit clause): A *unit clause* is a clause with a single literal.

Suppose (p) is a unit clause. Let $-p$ denote the negation of p, where double negation is collapsed (i.e., $-\neg q = q$). Then, the unit propagation rule is defined as follows:

- Assign the value of p to true.
- Remove all instances of $-p$ from clauses in the formula (shortening the corresponding clauses).
- Remove all clauses containing p (including the unit clause itself).

Example: Consider the formula $(A \vee B) \wedge (A \vee \neg B) \wedge (\neg A \vee B) \wedge (\neg A \vee \neg B) \wedge (A)$. The unit clause (A) is present in the formula. Applying the unit propagation rule, we remove all clauses containing A (positive literal), and remove $\neg A$ (negative literal) from the remaining clauses: ~~$(A \vee B)$~~ $\wedge$ ~~$(A \vee \neg B)$~~ $\wedge$ ~~$(\neg A \vee B)$~~ $\wedge$ ~~$(\neg A \vee B)$~~ $\wedge$ ~~$(\neg A \vee \neg B)$~~ $\wedge$ ~~(A)~~, which simplifies to $(B) \wedge (\neg B)$.

Pure Literal Rule

Definition 8 (Pure literal): A literal p is *pure* if it appears in the formula only positively or only negatively.

The pure literal rule is defined as follows:

- Assign the value of p to true.
- Remove all clauses containing a pure literal.

Example: Consider the formula $(A \vee B) \wedge (A \vee C) \wedge (B \vee C)$. The literal A is pure, as it appears only positively. Applying the pure literal rule, we assign $A = 1$ and remove all clauses containing A, which simplifies the formula to $(B \vee C)$.

Resolution Rule

1. Select a propositional variable p that appears both positively and negatively in the formula.
2. Partition the relevant clauses:
   - Let P be the set of all clauses that contain p.
   - Let N be the set of all clauses that contain $\neg p$.
3. Perform the resolution step:
   - For each pair of clauses $C_P \in P$ and $C_N \in N$, construct the *resolvent* by removing p and $\neg p$, then merging the remaining literals:

$$C_P \otimes_p C_N = (C_P \setminus \{p\}) \cup (C_N \setminus \{\neg p\})$$

   Example: $(a \vee b \vee \neg c) \otimes_b (a \vee \neg b \vee d \vee \neg e) = (a \vee \neg c \vee d \vee \neg e)$
4. Update the formula:
   - Remove all clauses in P and N.
   - Add the newly derived resolvents to the formula.

Davis–Putnam–Logemann–Loveland (DPLL) Algorithm

The DPLL algorithm [4] is a complete, backtracking search algorithm for deciding the satisfiability of propositional logic formulas in CNF, that is, for solving the CNF-SAT problem.

Introduced by Martin Davis, George Logemann, and Donald Loveland in 1961, the algorithm is a refinement of the Davis–Putnam algorithm.

In DPLL, the resolution rule is replaced with a *splitting* rule.

1. Let Δ be the current set of clauses.
2. Choose a propositional variable p occuring in the formula.
3. Test the satisfiability of $\Delta \cup \{(p)\}$.
   - If satisfiable, assign $p = 1$ and continue with the new formula.
   - If unsatisfiable, test the satisfiability of $\Delta \cup \{(\neg p)\}$.
     - If satisfiable, assign $p = 0$ and continue with the new formula.
     - If unsatisfiable, backtrack.

The DPLL algorithm is a *complete* algorithm: it will eventually find a satisfying assignment iff one exists.

DPLL Pseudocode

Input: set of clauses S
Output: *satisfiable* or *unsatisfiable*

```
1 S := propagate(S)
2 if S is empty then
    return satisfiable
3 if S contains the empty clause then
    return unsatisfiable
4 L := select_literal(S)
5 if DPLL(S ∪ {L}) = satisfiable then
    return satisfiable
6 else
    return DPLL(S ∪ {¬L})
7 end
```

§4 Advanced Topics

Abstract DPLL

Definition 9: Abstract DPLL is a high-level framework for a general and simple abstract rule-based formulation of the DPLL procedure. [5], [6]

DPLL procedure is being modelled by a *transition system*: a set of *states* and a *transition relation*.

- States are denoted by S.
- We write $S \Longrightarrow S'$ when the pair (S, S') is in the transition relation, meaning that S' is *reachable* from S in one *transition step*.
- We denote by $\Longrightarrow^*$ the reflexive-transitive closure of $\Longrightarrow$.
- We write $S \Longrightarrow^! S'$ if $S \Longrightarrow^* S'$ and S' is a *final* state, i.e., there is no S'' such that $S' \Longrightarrow S''$.
- A state is either *fail* or a pair $M \parallel F$, where M is a *model* (a sequence of *annotated literals*) and F is a finite set of clauses.
- An empty sequence of literals is denoted by $\emptyset$.
- A literal can be annotated as *decision literal*, which is denoted by l^d.
- We write F, C to denote the set $F \cup \{C\}$.

DPLL

The *basic DPLL system* consists of the following transition rules:

- *UnitPropagate*:
  $M \parallel F, (C \vee l) \Longrightarrow Ml \parallel F, (C \vee l)$ **if** $\begin{cases} M \vDash \neg C \\ l \text{ is undefined in } M \end{cases}$
- *PureLiteral*:
  $M \parallel F \Longrightarrow M \parallel F$ **if** $\begin{cases} l \text{ occurs in some clause of } F \\ \neg l \text{ does not occur in any clause of } F \\ l \text{ is undefined in } M \end{cases}$
- *Decide*:
  $M \parallel F, C \Longrightarrow Ml^d \parallel F, C$ **if** $\begin{cases} l \text{ or } \neg l \text{ occurs in a clause of } F \\ l \text{ is undefined in } M \end{cases}$
- *Fail*:
  $M \parallel F, C \Longrightarrow \mathit{fail}$ **if** $\begin{cases} M \vDash \neg C \\ M \text{ contains no decision literals} \end{cases}$
- *Backtrack*:
  $Ml^d N \parallel F, C \Longrightarrow M\neg l \parallel F, C$ **if** $\begin{cases} Ml^d N \vDash \neg C \\ N \text{ contains no decision literals} \end{cases}$

CDCL

Extended rules:

- *Learn*:
  $M \parallel F \Longrightarrow M \parallel F, C \quad \textbf{if} \begin{cases} \text{all atoms of } C \text{ occur in } F \\ F \models C \end{cases}$
- *Backjump*:
  $$Ml^d N \parallel F, C \Longrightarrow Ml' \parallel F, C \quad \textbf{if} \begin{cases} Ml^d N \models \neg C \\ \text{there is asome clause } C' \vee l' \text{ such that:} \\ F, C \models C' \vee l' \\ l' \text{ is undefined in } M \\ l' \vee \neg l' \text{ occurs in } F \text{ or in } Ml^d N \end{cases}$$
- *Forget*:
  $M \parallel F, C \Longrightarrow M \parallel F \quad \textbf{if} \{ F \models C$
- *Restart*:
  $M \parallel F \Longrightarrow \emptyset \parallel F$

TODO: discuss

TODO

- [ ] CDCL
- [◧] Abstract DPLL
- [ ] Encodings
- [ ] SAT Solvers
- [ ] Applications
- [ ] Exercises

Bibliography

[1] S. A. Cook, “The complexity of theorem-proving procedures,” in *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, 1971, pp. 151–158. doi: 10.1145/800157.805047.

[2] L. A. Levin, “Universal sequential search problems,” *Problemy Peredachi Informatsii*, vol. 9, no. 3, pp. 115–116, 1973, [Online]. Available: http://mi.mathnet.ru/ppi914

[3] M. Davis and H. Putnam, “A computing procedure for quantification theory,” *Journal of the ACM*, vol. 7, no. 3, pp. 201–215, 1960, doi: 10.1145/321033.321034.

[4] M. Davis, G. Logemann, and D. Loveland, “A machine program for theorem-proving,” *Communications of the ACM*, vol. 5, no. 7, pp. 394–397, 1962, doi: 10.1145/368273.368557.

[5] R. Nieuwenhuis, A. Oliveras, and C. Tinelli, “Abstract DPLL and Abstract DPLL Modulo Theories,” *Logic for Programming, Artificial Intelligence, and Reasoning*, vol. 3452. pp. 36–50, 2005. doi: 10.1007/978-3-540-32275-7_3.

Bibliography [2]

[6] R. Nieuwenhuis, A. Oliveras, and C. Tinelli, "Solving SAT and SAT Modulo Theories: From an abstract Davis-Putnam-Logemann-Loveland procedure to DPLL(T)," *Journal of the ACM*, vol. 53, no. 6, pp. 937–977, 2006, doi: 10.1145/1217856.1217859.