N7PD Declarative Programming

The SAT problem: theory and solvers

Rémi Delmas and Christophe Garion

ONERA/DTIM and ISAE-SUPAERO/DISC

License CC BY-NC-SA 3.0

Outline

1. **Introduction and basic definitions**
2. **Complexity**
3. **Resolution and the DPLL procedure**
4. **SAT applications**
5. **SAT implementation**

What is the SAT problem?

SAT is the abbreviation of the **Boolean Satisfiability Problem**: given a propositional formula, is there a propositional interpretation that satisfies it?

SAT is a important **theoretical** problem for CS: first problem to be proved to be NP-complete, phase transition…

What is the SAT problem?

SAT is the abbreviation of the **Boolean Satisfiability Problem**: given a propositional formula, is there a propositional interpretation that satisfies it?

SAT is a important **theoretical** problem for CS: first problem to be proved to be NP-complete, phase transition…

But SAT has also lots of **practical applications**:

- scheduling
- planning
- software verification
- tooling
- …

The propositional language

Definition (syntax of $\mathcal{L}_{PL}$)

Let *Var* be a set of propositional variables. The syntax of well-formed formulas of $\mathcal{L}_{PL}$ is given by the following EBNF:

$$\varphi ::= \text{'}A\text{'} \mid \text{'}\top\text{'} \mid \text{'}\bot\text{'} \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \varphi \rightarrow \varphi \mid \varphi \leftrightarrow \varphi$$

where $A \in Var$.

The semantics of propositional logic is defined classically (cf. OMI first year lecture).

Definition (interpretation)

An **interpretation** is a total function $Var \mapsto \{T, F\}$.

The **truth value** of a formula φ can be evaluated in an intepretation $\mathcal{I}$ using only the truth values of its subformulae.
The truth value of φ in $\mathcal{I}$ is denoted by $[\![\varphi]\!]_{\mathcal{I}}$

Interpretation, satisfiability, validity

Definition (interpretation)

An **interpretation** is a total function $Var \mapsto \{T, F\}$.

The **truth value** of a formula φ can be evaluated in an intepretation $\mathcal{I}$ using only the truth values of its subformulae.
The truth value of φ in $\mathcal{I}$ is denoted by $[\![\varphi]\!]_{\mathcal{I}}$

Definition (satisfiability)

A wff φ is **satisfiable** iff there is an interpretation $\mathcal{I}$ s.t. $[\![\varphi]\!]_{\mathcal{I}} = T$.

Definition (validity)

A wff φ is **valid** iff for all interpretations $\mathcal{I}$, $[\![\varphi]\!]_{\mathcal{I}} = T$. This is denoted by $\models \varphi$.

SAT and UNSAT

Notice that if a wff φ is **not satisfiable** (noted **UNSAT**), it means that $[\![\varphi]\!]_{\mathcal{I}} = F$ for **all interpretations** $\mathcal{I}$.

Thus, the following equivalence holds:

$$\varphi \text{ is valid} \Leftrightarrow \neg\varphi \text{ is UNSAT.}$$

Conjunctive Normal Form

We have seen in previous lectures that Conjunctive Normal Form (**CNF**) is a particular form of wff easy to manipulate.

Remember:

- a CNF is a **conjunction** of clauses
- a **clause** is a **disjunction** of literals
- a **literal** is either a prop. variable or the negation of a prop. variable

Conjunctive Normal Form

We have seen in previous lectures that Conjunctive Normal Form (**CNF**) is a particular form of wff easy to manipulate.

Remember:

- a CNF is a **conjunction** of clauses
- a **clause** is a **disjunction** of literals
- a **literal** is either a prop. variable or the negation of a prop. variable

Example:

$$\underbrace{(A \lor \neg B \lor D)}_{\text{a clause}} \land (\overbrace{\neg C \lor E}^{\text{literals}}) \land (\neg A \lor \neg C)$$

How to compute a CNF?

We have seen a simple algorithm to translate a wff φ into an equivalent CNF:

step 1 (remove impl.) $\begin{cases} \varphi \leftrightarrow \psi & \equiv (\varphi \rightarrow \psi) \wedge (\psi \rightarrow \varphi) \\ \varphi \rightarrow \psi & \equiv \neg\varphi \vee \psi \end{cases}$

step 2 (NNF) $\begin{cases} \neg(\neg\varphi) & \equiv \varphi \\ \neg(\varphi \wedge \psi) & \equiv \neg\varphi \vee \neg\psi \\ \neg(\varphi \vee \psi) & \equiv \neg\varphi \wedge \neg\psi \end{cases}$

step 3 (CNF) $\begin{cases} \varphi \vee (\psi \wedge \gamma) & \equiv (\varphi \vee \psi) \wedge (\varphi \vee \gamma) \\ (\psi \wedge \gamma) \vee \varphi & \equiv (\psi \vee \varphi) \wedge (\gamma \vee \varphi) \end{cases}$

How to compute a CNF?

We have seen a simple algorithm to translate a wff φ into an equivalent CNF:

step 1 (remove impl.) $\begin{cases} \varphi \leftrightarrow \psi & \equiv (\varphi \rightarrow \psi) \land (\psi \rightarrow \varphi) \\ \varphi \rightarrow \psi & \equiv \neg\varphi \lor \psi \end{cases}$

step 2 (NNF) $\begin{cases} \neg(\neg\varphi) & \equiv \varphi \\ \neg(\varphi \land \psi) & \equiv \neg\varphi \lor \neg\psi \\ \neg(\varphi \lor \psi) & \equiv \neg\varphi \land \neg\psi \end{cases}$

step 3 (CNF) $\begin{cases} \varphi \lor (\psi \land \gamma) & \equiv (\varphi \lor \psi) \land (\varphi \lor \gamma) \\ (\psi \land \gamma) \lor \varphi & \equiv (\psi \lor \varphi) \land (\gamma \lor \varphi) \end{cases}$

Problem

This algorithm may lead to an exponentially bigger formula!

Tseitin's transformation

Tseytin, G. S. (1970).
"On the complexity of derivation in propositional calculus".
In:
Studies in Constructive Mathematics and Mathematical Logic, part II.
Ed. by A. O. Slisenko.
Steklov Mathematical Institute,
Pp. 115–125.

Tseitin's transformation

Idea: start from the NNF and replace subformulas by new prop. variables.

step 3 (CNF)

$$\begin{cases} \varphi \vee \psi & \equiv (\varphi \vee \psi \vee \neg X) \wedge (\neg\varphi \vee X) \wedge (\neg\psi \vee X) \\ \varphi \wedge \psi & \equiv (\neg\varphi \vee \neg\psi \vee X) \wedge (\varphi \vee \neg X) \wedge (\psi \vee \neg X) \end{cases}$$

Tseitin's transformation

Idea: start from the NNF and replace subformulas by new prop. variables.

step 3 (CNF)
$$\begin{cases} \varphi \vee \psi \;\equiv (\varphi \vee \psi \vee \neg X) \wedge (\neg\varphi \vee X) \wedge (\neg\psi \vee X) \\ \varphi \wedge \psi \;\equiv (\neg\varphi \vee \neg\psi \vee X) \wedge (\varphi \vee \neg X) \wedge (\psi \vee \neg X) \end{cases}$$

Property (equisatisfiability of Tseitin's trans.)

The CNF obtained by Tseitin's transformation is equisatisfiable to the original wff.

Property

The size of the CNF obtained by Tseitin's transformation is linear w.r.t. the size of the original formula.

Tseitin's transformation

Idea: start from the NNF and replace subformulas by new prop. variables.

step 3 (CNF)
$$\begin{cases} \varphi \vee \psi & \equiv (\varphi \vee \psi \vee \neg X) \wedge (\neg \varphi \vee X) \wedge (\neg \psi \vee X) \\ \varphi \wedge \psi & \equiv (\neg \varphi \vee \neg \psi \vee X) \wedge (\varphi \vee \neg X) \wedge (\psi \vee \neg X) \end{cases}$$

Example:

Formula $\neg((A \wedge B) \vee C) \vee ((\neg A \vee C) \wedge (\neg C \vee A))$

CNF

Tseitin's transformation

Idea: start from the NNF and replace subformulas by new prop. variables.

step 3 (CNF) $$\begin{cases} \varphi \vee \psi & \equiv (\varphi \vee \psi \vee \neg X) \wedge (\neg\varphi \vee X) \wedge (\neg\psi \vee X) \\ \varphi \wedge \psi & \equiv (\neg\varphi \vee \neg\psi \vee X) \wedge (\varphi \vee \neg X) \wedge (\psi \vee \neg X) \end{cases}$$

Example:

Formula $\neg((\mathbf{A} \wedge \mathbf{B}) \vee C) \vee ((\neg A \vee C) \wedge (\neg C \vee A))$

CNF

Tseitin's transformation

Idea: start from the NNF and replace subformulas by new prop. variables.

step 3 (CNF)
$$\begin{cases} \varphi \lor \psi & \equiv (\varphi \lor \psi \lor \neg X) \land (\neg\varphi \lor X) \land (\neg\psi \lor X) \\ \varphi \land \psi & \equiv (\neg\varphi \lor \neg\psi \lor X) \land (\varphi \lor \neg X) \land (\psi \lor \neg X) \end{cases}$$

Example:

Formula $\neg(\mathbf{X_1} \lor C) \lor ((\neg A \lor C) \land (\neg C \lor A))$

CNF $(\neg A \lor \neg B \lor X_1) \land (A \lor \neg X_1) \land (B \lor \neg X_1)$

Tseitin's transformation

Idea: start from the NNF and replace subformulas by new prop. variables.

step 3 (CNF) $$\begin{cases} \varphi \vee \psi & \equiv (\varphi \vee \psi \vee \neg X) \wedge (\neg\varphi \vee X) \wedge (\neg\psi \vee X) \\ \varphi \wedge \psi & \equiv (\neg\varphi \vee \neg\psi \vee X) \wedge (\varphi \vee \neg X) \wedge (\psi \vee \neg X) \end{cases}$$

Example:

Formula $\neg(\mathbf{X_1} \vee \mathbf{C}) \vee ((\neg A \vee C) \wedge (\neg C \vee A))$

CNF $(\neg A \vee \neg B \vee X_1) \wedge (A \vee \neg X_1) \wedge (B \vee \neg X_1)$

Tseitin's transformation

Idea: start from the NNF and replace subformulas by new prop. variables.

step 3 (CNF)
$$\begin{cases} \varphi \vee \psi & \equiv (\varphi \vee \psi \vee \neg X) \wedge (\neg \varphi \vee X) \wedge (\neg \psi \vee X) \\ \varphi \wedge \psi & \equiv (\neg \varphi \vee \neg \psi \vee X) \wedge (\varphi \vee \neg X) \wedge (\psi \vee \neg X) \end{cases}$$

Example:

Formula $\neg(\mathbf{X_2}) \vee ((\neg A \vee C) \wedge (\neg C \vee A))$

CNF
$$(\neg A \vee \neg B \vee X_1) \wedge (A \vee \neg X_1) \wedge (B \vee \neg X_1) \wedge$$
$$(X_1 \vee C \vee \neg X_2) \wedge (\neg X_1 \vee X_2) \wedge (\neg C \vee X_2)$$

Tseitin's transformation

Idea: start from the NNF and replace subformulas by new prop. variables.

step 3 (CNF) $$\begin{cases} \varphi \vee \psi \ \equiv (\varphi \vee \psi \vee \neg X) \wedge (\neg\varphi \vee X) \wedge (\neg\psi \vee X) \\ \varphi \wedge \psi \ \equiv (\neg\varphi \vee \neg\psi \vee X) \wedge (\varphi \vee \neg X) \wedge (\psi \vee \neg X) \end{cases}$$

Example:

Formula $\neg(X_2) \vee ((\neg\mathbf{A} \vee \mathbf{C}) \wedge (\neg C \vee A))$

CNF $(\neg A \vee \neg B \vee X_1) \wedge (A \vee \neg X_1) \wedge (B \vee \neg X_1) \wedge$
$(X_1 \vee C \vee \neg X_2) \wedge (\neg X_1 \vee X_2) \wedge (\neg C \vee X_2)$

Tseitin's transformation

Idea: start from the NNF and replace subformulas by new prop. variables.

step 3 (CNF)
$$\begin{cases} \varphi \vee \psi \ \equiv (\varphi \vee \psi \vee \neg X) \wedge (\neg \varphi \vee X) \wedge (\neg \psi \vee X) \\ \varphi \wedge \psi \ \equiv (\neg \varphi \vee \neg \psi \vee X) \wedge (\varphi \vee \neg X) \wedge (\psi \vee \neg X) \end{cases}$$

Example:

Formula $\neg(X_2) \vee (\mathbf{X_3} \wedge (\neg C \vee A))$

CNF
$$\begin{aligned} &(\neg A \vee \neg B \vee X_1) \wedge (A \vee \neg X_1) \wedge (B \vee \neg X_1) \wedge \\ &(X_1 \vee C \vee \neg X_2) \wedge (\neg X_1 \vee X_2) \wedge (\neg C \vee X_2) \wedge \\ &(\neg A \vee C \vee \neg X_3) \wedge (A \vee X_3) \wedge (\neg C \vee X_3) \end{aligned}$$

Tseitin's transformation

Idea: start from the NNF and replace subformulas by new prop. variables.

step 3 (CNF) $\begin{cases} \varphi \vee \psi & \equiv (\varphi \vee \psi \vee \neg X) \wedge (\neg \varphi \vee X) \wedge (\neg \psi \vee X) \\ \varphi \wedge \psi & \equiv (\neg \varphi \vee \neg \psi \vee X) \wedge (\varphi \vee \neg X) \wedge (\psi \vee \neg X) \end{cases}$

Example:

Formula $\neg(X_2) \vee (X_3 \wedge (\neg \mathbf{C} \vee \mathbf{A}))$

CNF $(\neg A \vee \neg B \vee X_1) \wedge (A \vee \neg X_1) \wedge (B \vee \neg X_1) \wedge$
$(X_1 \vee C \vee \neg X_2) \wedge (\neg X_1 \vee X_2) \wedge (\neg C \vee X_2) \wedge$
$(\neg A \vee C \vee \neg X_3) \wedge (A \vee X_3) \wedge (\neg C \vee X_3)$

Tseitin's transformation

Idea: start from the NNF and replace subformulas by new prop. variables.

step 3 (CNF) $$\begin{cases} \varphi \lor \psi & \equiv (\varphi \lor \psi \lor \neg X) \land (\neg\varphi \lor X) \land (\neg\psi \lor X) \\ \varphi \land \psi & \equiv (\neg\varphi \lor \neg\psi \lor X) \land (\varphi \lor \neg X) \land (\psi \lor \neg X) \end{cases}$$

Example:

Formula $\neg(X_2) \lor (X_3 \land \mathbf{X_4})$

CNF
$$\begin{aligned} &(\neg A \lor \neg B \lor X_1) \land (A \lor \neg X_1) \land (B \lor \neg X_1) \land \\ &(X_1 \lor C \lor \neg X_2) \land (\neg X_1 \lor X_2) \land (\neg C \lor X_2) \land \\ &(\neg A \lor C \lor \neg X_3) \land (A \lor X_3) \land (\neg C \lor X_3) \land \\ &(\neg C \lor A \lor \neg X_4) \land (C \lor X_4) \land (\neg A \lor X_4) \end{aligned}$$

Tseitin's transformation

Idea: start from the NNF and replace subformulas by new prop. variables.

step 3 (CNF)
$$\begin{cases} \varphi \vee \psi & \equiv (\varphi \vee \psi \vee \neg X) \wedge (\neg\varphi \vee X) \wedge (\neg\psi \vee X) \\ \varphi \wedge \psi & \equiv (\neg\varphi \vee \neg\psi \vee X) \wedge (\varphi \vee \neg X) \wedge (\psi \vee \neg X) \end{cases}$$

Example:

Formula $\neg(X_2) \vee (\mathbf{X_3} \wedge \mathbf{X_4})$

CNF
$$\begin{aligned} &(\neg A \vee \neg B \vee X_1) \wedge (A \vee \neg X_1) \wedge (B \vee \neg X_1) \wedge \\ &(X_1 \vee C \vee \neg X_2) \wedge (\neg X_1 \vee X_2) \wedge (\neg C \vee X_2) \wedge \\ &(\neg A \vee C \vee \neg X_3) \wedge (A \vee X_3) \wedge (\neg C \vee X_3) \wedge \\ &(\neg C \vee A \vee \neg X_4) \wedge (C \vee X_4) \wedge (\neg A \vee X_4) \end{aligned}$$

Tseitin's transformation

Idea: start from the NNF and replace subformulas by new prop. variables.

step 3 (CNF) $\begin{cases} \varphi \lor \psi & \equiv (\varphi \lor \psi \lor \neg X) \land (\neg \varphi \lor X) \land (\neg \psi \lor X) \\ \varphi \land \psi & \equiv (\neg \varphi \lor \neg \psi \lor X) \land (\varphi \lor \neg X) \land (\psi \lor \neg X) \end{cases}$

Example:

Formula $\neg(X_2) \lor \mathbf{X_5}$

CNF
$$\begin{aligned}
&(\neg A \lor \neg B \lor X_1) \land (A \lor \neg X_1) \land (B \lor \neg X_1) \land \\
&(X_1 \lor C \lor \neg X_2) \land (\neg X_1 \lor X_2) \land (\neg C \lor X_2) \land \\
&(\neg A \lor C \lor \neg X_3) \land (A \lor X_3) \land (\neg C \lor X_3) \land \\
&(\neg C \lor A \lor \neg X_4) \land (C \lor X_4) \land (\neg A \lor X_4) \land \\
&(\neg X_3 \lor \neg X_4 \lor X_5) \land (X_3 \lor \neg X_5) \land (X_4 \lor \neg X_5)
\end{aligned}$$

Tseitin's transformation

Idea: start from the NNF and replace subformulas by new prop. variables.

step 3 (CNF) $\begin{cases} \varphi \vee \psi \equiv (\varphi \vee \psi \vee \neg X) \wedge (\neg\varphi \vee X) \wedge (\neg\psi \vee X) \\ \varphi \wedge \psi \equiv (\neg\varphi \vee \neg\psi \vee X) \wedge (\varphi \vee \neg X) \wedge (\psi \vee \neg X) \end{cases}$

Example:

Formula $\neg(\mathbf{X_2}) \vee \mathbf{X_5}$

CNF

$$\begin{aligned}
&(\neg A \vee \neg B \vee X_1) \wedge (A \vee \neg X_1) \wedge (B \vee \neg X_1) \wedge \\
&(X_1 \vee C \vee \neg X_2) \wedge (\neg X_1 \vee X_2) \wedge (\neg C \vee X_2) \wedge \\
&(\neg A \vee C \vee \neg X_3) \wedge (A \vee X_3) \wedge (\neg C \vee X_3) \wedge \\
&(\neg C \vee A \vee \neg X_4) \wedge (C \vee X_4) \wedge (\neg A \vee X_4) \wedge \\
&(\neg X_3 \vee \neg X_4 \vee X_5) \wedge (X_3 \vee \neg X_5) \wedge (X_4 \vee \neg X_5)
\end{aligned}$$

Tseitin's transformation

Idea: start from the NNF and replace subformulas by new prop. variables.

step 3 (CNF)
$$\begin{cases} \varphi \vee \psi \ \equiv (\varphi \vee \psi \vee \neg X) \wedge (\neg\varphi \vee X) \wedge (\neg\psi \vee X) \\ \varphi \wedge \psi \ \equiv (\neg\varphi \vee \neg\psi \vee X) \wedge (\varphi \vee \neg X) \wedge (\psi \vee \neg X) \end{cases}$$

Example:

Formula $\mathbf{X_6}$

CNF
$$\begin{array}{l}
(\neg A \vee \neg B \vee X_1) \wedge (A \vee \neg X_1) \wedge (B \vee \neg X_1) \wedge \\
(X_1 \vee C \vee \neg X_2) \wedge (\neg X_1 \vee X_2) \wedge (\neg C \vee X_2) \wedge \\
(\neg A \vee C \vee \neg X_3) \wedge (A \vee X_3) \wedge (\neg C \vee X_3) \wedge \\
(\neg C \vee A \vee \neg X_4) \wedge (C \vee X_4) \wedge (\neg A \vee X_4) \wedge \\
(\neg X_3 \vee \neg X_4 \vee X_5) \wedge (X_3 \vee \neg X_5) \wedge (X_4 \vee \neg X_5) \wedge \\
(\neg X_2 \vee X_5 \vee \neg X_6) \wedge (X_2 \vee X_6) \wedge (\neg X_5 \vee X_6)
\end{array}$$

Using CNF

As there is a linear transformation of wff to CNF, **we will restrict our presentation on SAT to CNF**.

For readability, we will denote a CNF by a set of clauses, each clause being denoted by a set of literals.

For instance:

$$(x_1 \vee \neg x_2) \wedge (x_3 \vee x_4 \vee x_5) \wedge (x_5 \vee \neg x_2)$$
$$\equiv$$
$$\{\{x_1, \bar{x_2}\}, \{x_3, x_4, x_5\}, \{x_5, \bar{x_2}\}\}$$

Outline

Is SAT a complex problem?

We might wonder first if SAT is **really** a complex problem.

Is SAT a complex problem?

We might wonder first if SAT is **really** a complex problem.

First, let us look at the simpliest algorithm to check if a wff is SAT or not: **truth tables**.

Truth table complexity is clearly $O(2^n)$ where n is the number of propositional variables involved in the wff.

OK, but is there a better algorithm for SAT?

Is SAT a complex problem?

We might wonder first if SAT is **really** a complex problem.

First, let us look at the simpliest algorithm to check if a wff is SAT or not: **truth tables**.

Truth table complexity is clearly $O(2^n)$ where n is the number of propositional variables involved in the wff.

OK, but is there a better algorithm for SAT?

The answer is: **we do not know!**

But if there is an efficient (i.e. polynomial) algorithm for SAT, then it would solve the $P = NP$ question...

Is SAT a complex problem?

We might wonder first if SAT is **really** a complex problem.

First, let us look at the simpliest algorithm to check if a wff is SAT or not: **truth tables**.

Truth table complexity is clearly $O(2^n)$ where n is the number of propositional variables involved in the wff.

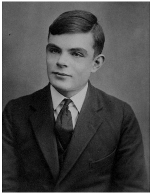
OK, but is there a better algorithm for SAT?

The answer is: **we do not know!**

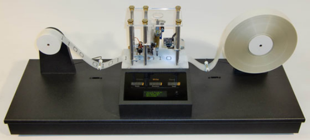
But if there is an efficient (i.e. polynomial) algorithm for SAT, then it would solve the $P = NP$ question...

In order to explain that, let us go back to the basics of computation: Turing machines.

A mathematical model for computation: the Turing machine



The Turing machine (Turing, 1936)



This is the theoretical foundation of computers and **imperative programming**:

- tape: memory with stored program
- automaton: microprocessor

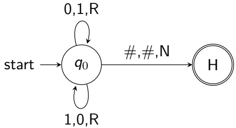
École Normale Supérieure de Lyon (2015).
The RubENS project.
http://rubens.ens-lyon.fr/.

The Turing machine: how does it work?

The Turing machine can be viewed as an **automaton** with an **infinite tape** and a **read/write head**.

- the automaton is composed of **states**
- states can be linked by **transitions**
- transitions are **triggered** by the symbol read by the head
- during a transition, the head may **write** a new symbol and **move** on the new left/right position

current state	current symbol	next state	symbol to write	move head
q_0	0	q_0	1	RIGHT
q_0	1	q_0	0	RIGHT
q_0	#	HALT	#	NONE



Again, see OMI first year lecture, part on languages.

Nondeterministic Turing machines

A **Nondeterministic Turing machine** (NTM) is a Turing machine for which transitions are no more represented by functions, but rather by **relations**.

This means that for each pair (state, symbol), there may be **more than one** tuple (state, symbol, action) or none.

Nondeterministic Turing machines

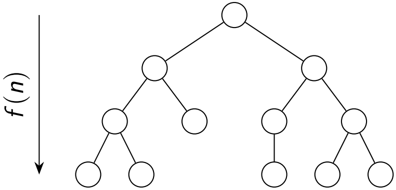
A **Nondeterministic Turing machine** (NTM) is a Turing machine for which transitions are no more represented by functions, but rather by **relations**.

This means that for each pair (state, symbol), there may be **more than one** tuple (state, symbol, action) or none.

A NTM will accept an input if there is **some** sequence of **nondeterministic** choices that results in YES.

Nondeterministic Turing machines

Intuitively, a NTM decides a language L in time $f(n)$ where n is the size of the input if $f(n)$ bounds the height of the "computational tree" induced by the machine:



For the moment, there is **no polynomial reduction** of NTM to (deterministic) Turing machine, only exponential ones!

NP problems

NP is a class of problems that can decided in **polynomial time** by a NTM.

Intuitively, a problem in NP is such that verifying if an alternative is a solution is polynomial.

For instance, SAT is in NP: given an interpretation, you can check that the truth value of input formula is T in polynomial time...
... but we do not know if SAT is in P.

NP problems

NP is a class of problems that can decided in **polynomial time** by a NTM.

Intuitively, a problem in NP is such that verifying if an alternative is a solution is polynomial.

For instance, SAT is in NP: given an interpretation, you can check that the truth value of input formula is T in polynomial time...
... but we do not know if SAT is in P.

Of course, $P \subseteq NP$, but we do not know if $NP \subseteq P$ (this is a 1,000,000$ prize ☺).

NP-complete problems

A **NP-complete** problem is a problem that:

- is in NP
- is such that every problem in NP can be reduced to the problem in polynomial time

NP-complete problems

A **NP-complete** problem is a problem that:

- is in NP
- is such that every problem in NP can be reduced to the problem in polynomial time

Therefore:

- showing that a NP-complete problem is in P implies that **all problems in NP are in P**
- showing that a problem is in NPC gives a good indication on the fact that the problem should be the least likely to be in P...

SAT is NP-complete

Theorem (Cook-Levin)

SAT is NP-complete.

Cook, Stephen (1971).
"The complexity of theorem proving procedures".
In: **Proc. of the third annual ACM symposium on Theory of computing** ,
Pp. 151–158.
http://4mhz.de/cook.html.

SAT is NP-complete

Theorem (Cook-Levin)

SAT is NP-complete.

Cook, Stephen (1971).
"The complexity of theorem proving procedures".
In: **Proc. of the third annual ACM symposium on Theory of computing** ,
Pp. 151–158.
http://4mhz.de/cook.html.

Idea: given a NP problem $\mathcal{P}$, a NTM $\mathcal{M}$ that solves it, and an entry $\mathcal{I}$ for $\mathcal{P}$, build a formula that is satisfiable iff $\mathcal{M}$ accepts $\mathcal{I}$ for $\mathcal{P}$.

SAT variants

There are restricted versions of SAT:

Problem	Complexity
CNFSAT	NP-complete
2SAT	P
3SAT	NP-complete
HORNSAT	P

In the following, we will use CNFSAT instead of SAT.

Outline

Outline

Back to Resolution

We want to show that a set of clauses is UNSAT.

➥ we could use Resolution to do that!

Back to Resolution

We want to show that a set of clauses is UNSAT.

➥ we could use Resolution to do that!

Remember:

- Resolution has only two rules:

$$\frac{\{\mathbf{l}, l_1, \ldots, l_n\} \qquad \{\bar{\mathbf{l}}, l'_1, \ldots, l'_m\}}{\{l_1, \ldots, l_n, l'_1, \ldots, l'_m\}}\ (R) \qquad \frac{\{\mathbf{l}, \mathbf{l}, l_1, \ldots, l_n\}}{\{l, l_1, \ldots, l_n\}}\ (F)$$

- we can also add a **subsumption** rule:

$$\frac{\{\mathbf{l_1}, \ldots, \mathbf{l_n}\} \qquad \{\mathbf{l_1}, \ldots, \mathbf{l_n}, l'_1, \ldots, l'_m\}}{\{l_1, \ldots, l_n\}}\ (S)$$

- Resolution is sound and complete for **refutation**

How to **efficiently** use Resolution?

Resolution: saturation method

A first idea: apply Resolution again and again (there is a fixed point!).

Resolution: saturation method

Algorithm 3.1: Resolution by saturation algorithm

Input: a set of clauses S
Output: SAT if S is satisfiable, UNSAT else

$S' \leftarrow \emptyset$;
while $S' \neq S$ **do**
    $S' \leftarrow S$;
    generate **all** possible resolvent and stores them in R
    foreach resolvent r **do**
        if $r = \square$ **then**
            return UNSAT;
        end
        if there is no clause C such that C subsumes r **then**
            remove all clauses subsumed by r ;
            $S = S \cup \{r\}$;
        end
    end
end
return SAT;

Resolution: saturation method

But

- huge memory consumption: intermediate resolvent are computed even if they are subsumed by another clause or are tautologies
- how to efficiently find the clauses on which to apply Resolution
- in which order should you apply Resolution to optimize the process?

The Davis and Putnam procedure

Davis and Putnam developed in 1960 an algorithm using Resolution to check if a set of clauses is SAT or UNSAT.

Davis, Martin and Hilary Putnam (July 1960).
"A Computing Procedure for Quantification Theory".
In: **Journal of the ACM** 7.3,
Pp. 201–215.
ISSN: 0004-5411.
DOI: 10.1145/321033.321034.
http://doi.acm.org/10.1145/321033.321034.

The Davis and Putnam procedure

Davis and Putnam developed in 1960 an algorithm using Resolution to check if a set of clauses is SAT or UNSAT.

Three rules are used in the procedure:

taut a clause $\{\mathbf{x}, \bar{\mathbf{x}}, l_1, \ldots, l_n\}$ can be eliminated

pure if a variable appears **exclusively** as a positive (or negative) literal l , then all the clauses containing l can be eliminated

res(x) that applies Resolution rule using variable x:

- for each pair of clauses $\{x, l_1, \ldots, l_n\}$ and $\{\bar{x}, l'_1, \ldots, l'_m\}$ generate $\{l_1, \ldots, l_n, l'_1, \ldots, l'_m\}$
- remove all clauses containing either x or $\bar{x}$

The Davis and Putnam procedure

Davis and Putnam developed in 1960 an algorithm using Resolution to check if a set of clauses is SAT or UNSAT.

Algorithm 3.2: Davis and Putnam algorithm

Input: a set of clauses S
Output: SAT if S is satisfiable, UNSAT else

$S' \leftarrow \emptyset$;
while $S' \neq S$ **do**
    $S' \leftarrow S$;
    apply `taut` rule ;
    apply `unit` rule ;
    apply `pure` rule ;
    choose a variable x and apply `res` using x ;
end
if $S = \emptyset$ **then**
    return UNSAT;
else
    return SAT;
end

DP procedure: example

init.		$\{\{x_1\bar{x_2}\bar{x_3}\}\{\bar{x_1}\bar{x_2}\bar{x_3}\}\{x_2x_3\}\{x_3x_4\}\{x_3\bar{x_4}\}\{\bar{x_3}\}\{\mathbf{x_5}\}\{\bar{x_5}x_4\}\}$
`pure` on x_5.	$\rightsquigarrow$	$\{\{\mathbf{x_1}\bar{x_2}\bar{x_3}\}\{\bar{\mathbf{x_1}}\bar{x_2}\bar{x_3}\}\{x_2x_3\}\{x_3x_4\}\{x_3\bar{x_4}\}\{\bar{x_3}\}\}$
`res` on x_1	$\rightsquigarrow$	$\{\{\bar{\mathbf{x_2}}\bar{x_3}\}\{\mathbf{x_2}x_3\}\{x_3x_4\}\{x_3\bar{x_4}\}\{\bar{x_3}\}\}$
`res` on x_2	$\rightsquigarrow$	$\{\{\mathbf{x_3}\bar{\mathbf{x_3}}\}\{x_3x_4\}\{x_3\bar{x_4}\}\{\bar{x_3}\}\}$
`taut`	$\rightsquigarrow$	$\{\{x_3\mathbf{x_4}\}\{x_3\bar{\mathbf{x_4}}\}\{\bar{x_3}\}\}$
`res` on x_4	$\rightsquigarrow$	$\{\{\mathbf{x_3}\}\{\bar{\mathbf{x_3}}\}\}$
`res` on x_3	$\rightsquigarrow$	$\emptyset$
`UNSAT`		

DP procedure: example

init.		$\{\{x_1\bar{x_2}\bar{x_3}\}\{\bar{x_1}\bar{x_2}\bar{x_3}\}\{x_2x_3\}\{x_3x_4\}\{x_3\bar{x_4}\}\{\bar{x_3}\}\{\mathbf{x_5}\}\{\bar{x_5}x_4\}\}$
`pure` on x_5.	$\leadsto$	$\{\{\mathbf{x_1}\bar{x_2}\bar{x_3}\}\{\mathbf{\bar{x_1}}\bar{x_2}\bar{x_3}\}\{x_2x_3\}\{x_3x_4\}\{x_3\bar{x_4}\}\{\bar{x_3}\}\}$
`res` on x_1	$\leadsto$	$\{\{\mathbf{\bar{x_2}}\bar{x_3}\}\{\mathbf{x_2}x_3\}\{x_3x_4\}\{x_3\bar{x_4}\}\{\bar{x_3}\}\}$
`res` on x_2	$\leadsto$	$\{\{\mathbf{x_3\bar{x_3}}\}\{x_3x_4\}\{x_3\bar{x_4}\}\{\bar{x_3}\}\}$
`taut`	$\leadsto$	$\{\{x_3\mathbf{x_4}\}\{x_3\mathbf{\bar{x_4}}\}\{\bar{x_3}\}\}$
`res` on x_4	$\leadsto$	$\{\{\mathbf{x_3}\}\{\mathbf{\bar{x_3}}\}\}$
`res` on x_3	$\leadsto$	$\emptyset$
`UNSAT`		

In practise, DP is **not efficient**:

- takes a lot of time to find clauses on which `res` can be applied
- memory is saturated with the computed resolvents

Outline

DPPL principles

The DPPL procedure is a refinement of the DP procedure.

Davis, Martin, Georges Logemann, and Donald Loveland (1962). "A machine program for theorem proving". In: **Communications of the ACM** 5.7, Pp. 394–397. DOI: 10.1145/368273.368557.

DPPL principles

The DPPL procedure is a refinement of the DP procedure.

> Davis, Martin, Georges Logemann, and Donald Loveland (1962). "A machine program for theorem proving". In: **Communications of the ACM** 5.7, Pp. 394–397. DOI: 10.1145/368273.368557.

The principles of the procedure are the following:

- an interpretation/a model M satisfying the formula φ is built incrementally
- M is extended
  - either by **deduction** of the value of a literal l from M and φ
  - either by **deciding** the value of a literal l that does not appear in M
- if a decision leads to a failure node (a clause is valuated to F), the algorithm **backtracks** and inverse the decision

DPLL: example

Operation	*Model*	Formula
		$\{x_1, x_2\}, \{x_2, \bar{x_3}, x_4\}, \{\bar{x_1}, \bar{x_2}\}, \{\bar{x_1}, \bar{x_3}, \bar{x_4}\}, \{x_1\}$

DPLL: example

Operation	*Model*	Formula
		$\{x_1, x_2\}, \{x_2, \bar{x_3}, x_4\}, \{\bar{x_1}, \bar{x_2}\}, \{\bar{x_1}, \bar{x_3}, \bar{x_4}\}, \{x_1\}$
Deduce x_1	x_1	$\{x_1, x_2\}, \{x_2, \bar{x_3}, x_4\}, \{\bar{x_1}, \bar{x_2}\}, \{\bar{x_1}, \bar{x_3}, \bar{x_4}\}, \{x_1\}$

DPLL: example

Operation	*Model*	Formula
		$\{x_1, x_2\}, \{x_2, \bar{x_3}, x_4\}, \{\bar{x_1}, \bar{x_2}\}, \{\bar{x_1}, \bar{x_3}, \bar{x_4}\}, \{x_1\}$
Deduce x_1	x_1	$\{x_1, x_2\}, \{x_2, \bar{x_3}, x_4\}, \{\bar{x_1}, \bar{x_2}\}, \{\bar{x_1}, \bar{x_3}, \bar{x_4}\}, \{x_1\}$
Deduce x_2	x_1, x_2	$\{x_1, x_2\}, \{x_2, \bar{x_3}, x_4\}, \{\bar{x_1}, \bar{x_2}\}, \{\bar{x_1}, \bar{x_3}, \bar{x_4}\}, \{x_1\}$

DPLL: example

Operation	*Model*	Formula
		$\{x_1, x_2\}, \{x_2, \bar{x_3}, x_4\}, \{\bar{x_1}, \bar{x_2}\}, \{\bar{x_1}, \bar{x_3}, \bar{x_4}\}, \{x_1\}$
Deduce x_1	x_1	$\{x_1, x_2\}, \{x_2, \bar{x_3}, x_4\}, \{\bar{x_1}, \bar{x_2}\}, \{\bar{x_1}, \bar{x_3}, \bar{x_4}\}, \{x_1\}$
Deduce x_2	x_1, x_2	$\{x_1, x_2\}, \{x_2, \bar{x_3}, x_4\}, \{\bar{x_1}, \bar{x_2}\}, \{\bar{x_1}, \bar{x_3}, \bar{x_4}\}, \{x_1\}$
Decide x_3	x_1, x_2, x_3	$\{x_1, x_2\}, \{x_2, \bar{x_3}, x_4\}, \{\bar{x_1}, \bar{x_2}\}, \{\bar{x_1}, \bar{x_3}, \bar{x_4}\}, \{x_1\}$

DPLL: example

Operation	*Model*	Formula
		$\{x_1, x_2\}, \{x_2, \bar{x_3}, x_4\}, \{\bar{x_1}, \bar{x_2}\}, \{\bar{x_1}, \bar{x_3}, \bar{x_4}\}, \{x_1\}$
Deduce x_1	x_1	$\{x_1, x_2\}, \{x_2, \bar{x_3}, x_4\}, \{\bar{x_1}, \bar{x_2}\}, \{\bar{x_1}, \bar{x_3}, \bar{x_4}\}, \{x_1\}$
Deduce x_2	x_1, x_2	$\{x_1, x_2\}, \{x_2, \bar{x_3}, x_4\}, \{\bar{x_1}, \bar{x_2}\}, \{\bar{x_1}, \bar{x_3}, \bar{x_4}\}, \{x_1\}$
Decide x_3	x_1, x_2, x_3	$\{x_1, x_2\}, \{x_2, \bar{x_3}, x_4\}, \{\bar{x_1}, \bar{x_2}\}, \{\bar{x_1}, \bar{x_3}, \bar{x_4}\}, \{x_1\}$
Deduce x_4	x_1, x_2, x_3, x_4	$\{x_1, x_2\}, \{x_2, \bar{x_3}, x_4\}, \{\bar{x_1}, \bar{x_2}\}, \{\bar{x_1}, \bar{x_3}, \bar{x_4}\}, \{x_1\}$

DPLL: example

Operation	*Model*	Formula
		$\{x_1, x_2\}, \{x_2, \bar{x_3}, x_4\}, \{\bar{x_1}, \bar{x_2}\}, \{\bar{x_1}, \bar{x_3}, \bar{x_4}\}, \{x_1\}$
Deduce x_1	x_1	$\{x_1, x_2\}, \{x_2, \bar{x_3}, x_4\}, \{\bar{x_1}, \bar{x_2}\}, \{\bar{x_1}, \bar{x_3}, \bar{x_4}\}, \{x_1\}$
Deduce x_2	x_1, x_2	$\{x_1, x_2\}, \{x_2, \bar{x_3}, x_4\}, \{\bar{x_1}, \bar{x_2}\}, \{\bar{x_1}, \bar{x_3}, \bar{x_4}\}, \{x_1\}$
Decide x_3	x_1, x_2, x_3	$\{x_1, x_2\}, \{x_2, \bar{x_3}, x_4\}, \{\bar{x_1}, \bar{x_2}\}, \{\bar{x_1}, \bar{x_3}, \bar{x_4}\}, \{x_1\}$
Deduce x_4	x_1, x_2, x_3, x_4	$\{x_1, x_2\}, \{x_2, \bar{x_3}, x_4\}, \{\bar{x_1}, \bar{x_2}\}, \{\bar{x_1}, \bar{x_3}, \bar{x_4}\}, \{x_1\}$
Undo x_3	x_1, x_2	$\{x_1, x_2\}, \{x_2, \bar{x_3}, x_4\}, \{\bar{x_1}, \bar{x_2}\}, \{\bar{x_1}, \bar{x_3}, \bar{x_4}\}, \{x_1\}$

DPLL: example

Operation	*Model*	Formula
		$\{x_1, x_2\}, \{x_2, \bar{x}_3, x_4\}, \{\bar{x}_1, \bar{x}_2\}, \{\bar{x}_1, \bar{x}_3, \bar{x}_4\}, \{x_1\}$
Deduce x_1	x_1	$\{x_1, x_2\}, \{x_2, \bar{x}_3, x_4\}, \{\bar{x}_1, \bar{x}_2\}, \{\bar{x}_1, \bar{x}_3, \bar{x}_4\}, \{x_1\}$
Deduce x_2	x_1, x_2	$\{x_1, x_2\}, \{x_2, \bar{x}_3, x_4\}, \{\bar{x}_1, \bar{x}_2\}, \{\bar{x}_1, \bar{x}_3, \bar{x}_4\}, \{x_1\}$
Decide x_3	x_1, x_2, x_3	$\{x_1, x_2\}, \{x_2, \bar{x}_3, x_4\}, \{\bar{x}_1, \bar{x}_2\}, \{\bar{x}_1, \bar{x}_3, \bar{x}_4\}, \{x_1\}$
Deduce x_4	x_1, x_2, x_3, x_4	$\{x_1, x_2\}, \{x_2, \bar{x}_3, x_4\}, \{\bar{x}_1, \bar{x}_2\}, \{\bar{x}_1, \bar{x}_3, \bar{x}_4\}, \{x_1\}$
Undo x_3	x_1, x_2	$\{x_1, x_2\}, \{x_2, \bar{x}_3, x_4\}, \{\bar{x}_1, \bar{x}_2\}, \{\bar{x}_1, \bar{x}_3, \bar{x}_4\}, \{x_1\}$
Decide x_3	x_1, x_2, x_3	$\{x_1, x_2\}, \{x_2, \bar{x}_3, x_4\}, \{\bar{x}_1, \bar{x}_2\}, \{\bar{x}_1, \bar{x}_3, \bar{x}_4\}, \{x_1\}$

SAT: an abstract framework

In Nieuwenhuis, Oliveras, and Tinelli 2004, Nieuwenhuis et al. give an abstract framework for SAT algorithms.

Nieuwenhuis, R., A. Oliveras, and C. Tinelli (2004).
"Abstract DPLL and Abstract DPLL Modulo Theories".
In: **LPAR** ,
Pp. 36–50.

This framework is based on an **abstract transition system**.

Abstract SAT: states

Definition (states)

A state in the transition system is either

- *fail*
- $M \| \varphi$ where
  - φ is a CNF
  - M is a (partial) interpretation $l_1, \ldots, l_i^d, \ldots, l_k$ where the l_i are literals and d a **decision level**

Definition (states)

A state in the transition system is either

- *fail*
- $M \| \varphi$ where
  - φ is a CNF
  - M is a (partial) interpretation $l_1, \ldots, l_i^d, \ldots, l_k$ where the l_i are literals and d a **decision level**

Definition (initial and final states)

- initial state: $\emptyset \| \varphi$
- final states:
  - *fail* if φ is UNSAT
  - $M \| \psi$ where ψ is equivalent to φ and M is a model of ψ

How does DPPL work?

DPLL builds incrementally a model for the initial formula.

During the space exploration:

- a variable may have for value T (true), F (false) or X (unassigned)
- a clause can be

sat	iff one of its literals is T
unit	iff all of its literals are F except one
conflict	iff all of its literals are F
undef	otherwise

- a CNF is **SAT** iff all its clauses are sat

DPLL: model extension rules

Rule (UnitProp)

$$M \| \varphi, C \vee l \rightarrow M\ l \| \varphi, C \vee l \text{ if } \begin{cases} M \models \neg C \\ l \text{ is not defined in } M \end{cases}$$

Rule (UnitProp)

$$M \| \varphi, C \vee l \rightarrow M\ l \| \varphi, C \vee l \text{ if } \begin{cases} M \models \neg C \\ l \text{ is not defined in } M \end{cases}$$

Rule (Decide)

$$M \| \varphi \rightarrow M\ l^d \| \varphi \text{ if } \begin{cases} l \text{ or } \overline{l} \text{ is in } \varphi \\ l \text{ is not defined in } \ M \end{cases}$$

l^d is identified as a literal of decision level d.

DPLL: model repairing rules

Rule (Fail)

$$M \| \varphi', C \to \mathit{fail} \text{ if } \begin{cases} M \models \neg C \\ M \text{ has no decision literal} \end{cases}$$

DPLL: model repairing rules

Rule (Fail)

$$M \| \varphi', C \to \mathit{fail} \text{ if } \begin{cases} M \models \neg C \\ M \text{ has no decision literal} \end{cases}$$

Rule (Backtrack)

$$M\ l^d\ N \| \varphi', C \to M\ \bar{l} \| \varphi', C \text{ if } \begin{cases} M\ l^d\ N \models \neg C \\ l^d \text{ is the last decision literal} \end{cases}$$

DPLL basic procedure

Definition (DPLL basic procedure)

The original DPLL algorithm from Davis, Logemann, and Loveland 1962 uses the following rules:

- UnitProp
- Decide
- Fail
- Backtrack

until a model or *fail* is produced.

Correction of the basic DPLL procedure

Some definitions:

Irreductible state	state from which no transition can be fired
Execution	sequence of transitions allowed by the rules from an initial state $\emptyset \| \varphi$
Saturated execution	execution finishing with an irreductible state

Correction of the basic DPLL procedure

Some definitions:

Irreductible state	state from which no transition can be fired
Execution	sequence of transitions allowed by the rules from an initial state $\emptyset \| \varphi$
Saturated execution	execution finishing with an irreductible state

Theorem (strong termination)

*Every execution of DPPL is **finite**.*

Correction of the basic DPLL procedure

Some definitions:

Irreductible state	state from which no transition can be fired
Execution	sequence of transitions allowed by the rules from an initial state $\emptyset \| \varphi$
Saturated execution	execution finishing with an irreductible state

Theorem (strong termination)

*Every execution of DPPL is **finite**.*

Theorem (soundness)

For each saturated execution from $\emptyset \| \varphi$ finishing in $M \| \varphi$, $M \models \varphi$.

Correction of the basic DPLL procedure

Some definitions:

Irreductible state	state from which no transition can be fired
Execution	sequence of transitions allowed by the rules from an initial state $\emptyset \| \varphi$
Saturated execution	execution finishing with an irreductible state

Theorem (strong termination)

*Every execution of DPPL is **finite**.*

Theorem (soundness)

For each saturated execution from $\emptyset \| \varphi$ finishing in $M \| \varphi$, $M \models \varphi$.

Theorem (completeness)

If φ is UNSAT, every saturated execution from $\emptyset \| \varphi$ finishes in fail.

Outline

DPLL: from backtracking to backjumping

Rule (Backtrack)

$$M\ l^d\ N \| \varphi', C \to M\ \bar{l} \| \varphi', C \text{ if } \begin{cases} M\ l^d\ N \models \neg C \\ l^d \text{ is the last decision literal} \end{cases}$$

DPLL: from backtracking to backjumping

Rule (Backtrack)

$$M\ l^d\ N \| \varphi', C \to M\ \bar{l} \| \varphi', C \text{ if } \begin{cases} M\ l^d\ N \models \neg C \\ l^d \text{ is the last decision literal} \end{cases}$$

Rule (Backjump)

$$M\ l^d\ N \| \varphi', C \to M\ k \| \varphi', C \text{ if } \begin{cases} 1.\ M\ l^d\ N \models \neg C \\ 2.\ \text{there is a clause } D \vee k \text{ s.t.} \\ \quad \varphi', C \models D \vee k \text{ and } M \models \neg D \\ \quad k \text{ is not defined in M} \\ \quad k \text{ or } \bar{k} \text{ occurs in } \varphi' \vee C \end{cases}$$

CDCL: learning from errors

How to avoid repeating the same errors when encountering a *fail* branch?

CDCL: learning from errors

How to avoid repeating the same errors when encountering a *fail* branch?

Idea: find the clause “responsible” for the conflict, add it and backjump to the highest decision level.
Adding the clause avoids getting in the same subtree if encountering the same subinterpretation.

This technics is called Conflict-Driven Clause Learning (CDCL).

Modern DPLL: new rules

Rule (Learn)

$$M \| \varphi \to M \| \varphi, C \text{ if } \begin{cases} \text{all atoms of } C \text{ appear in } \varphi \\ \varphi \models C \end{cases}$$

Rule (Learn)

$$M \| \varphi \rightarrow M \| \varphi, C \text{ if } \begin{cases} \text{all atoms of } C \text{ appear in } \varphi \\ \varphi \models C \end{cases}$$

Rule (Forget)

$$M \| \varphi, C \rightarrow M \| \varphi \text{ if } \varphi \models C$$

Modern DPLL: new rules

Rule (Learn)

$$M \| \varphi \to M \| \varphi, C \text{ if } \begin{cases} \text{all atoms of } C \text{ appear in } \varphi \\ \varphi \models C \end{cases}$$

Rule (Forget)

$$M \| \varphi, C \to M \| \varphi \text{ if } \varphi \models C$$

Rule (Restart)

$$M \| \varphi \to \emptyset \| \varphi \text{ if you want...}$$

Strategies for modern DPLL

- applying a rule of basic DPPL between each Learn and applying Restart less and less often maintains termination.
- in practise, Learn is applied just after each Backjump
- the most common strategy uses these priorities:
  1. if $n > 0$ conflicts have been found, then increment n and apply Restart
  2. if a clause is falsified by the current interpretation, apply Fail or Backjump + Learn
  3. apply UnitProp until saturation

Modern DPLL: correctness

Theorem (termination)

Every execution in which

- *Learn/Forget are applied a fixed number of times*
- *Restart is applied with a growing periodicity*

is finite.

Modern DPLL: correctness

Theorem (termination)

Every execution in which

- *Learn/Forget are applied a fixed number of times*
- *Restart is applied with a growing periodicity*

is finite.

Theorem (soundness)

For each saturated execution from $\emptyset \| \varphi$ *finishing in* $M \| \varphi$, $M \models \varphi$.

Modern DPLL: correctness

Theorem (termination)

Every execution in which

- *Learn/Forget are applied a fixed number of times*
- *Restart is applied with a growing periodicity*

is finite.

Theorem (soundness)

For each saturated execution from $\emptyset \| \varphi$ *finishing in* $M \| \varphi$, $M \models \varphi$.

Theorem (completeness)

If φ *is UNSAT, every saturated execution from* $\emptyset \| \varphi$ *finishes in fail.*

Actual solvers

Current solvers combine

- a **preprocessing** phase using Resolution to simplify the initial formula before using DPLL
- a **parallel** implementation of modern DPLL
- **inprocessing** phases using Resolution to simplify Resolution the formula during DPLL

Actual solvers

Current solvers combine

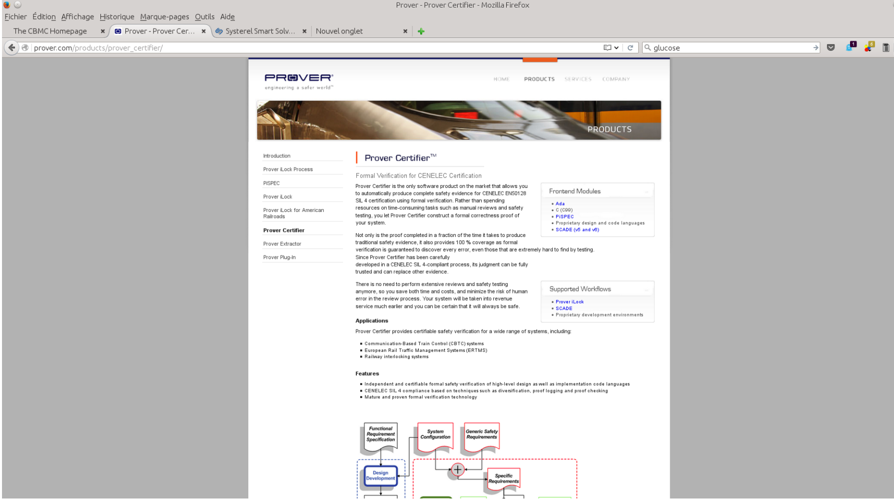
- a **preprocessing** phase using Resolution to simplify the initial formula before using DPLL
- a **parallel** implementation of modern DPLL
- **inprocessing** phases using Resolution to simplify Resolution the formula during DPLL

Some interesting solvers:

- Chaff (2000), minisat (2004), PicoSAT (2006)
- SAT4J (written in Java, integrated in Eclipse for libraries dependency)
- Glucose 4.0 (written in C++, winner of several contests)
- Lingeling (written in C, winner of several contests)

Outline

Prover Technology



Scade Design Verifier

SCADE Suite Design Verifier | Esterel Technologies - Mozilla Firefox

Fichier Édition Affichage Historique Marque-pages Outils Aide

The CBMC Homepage | Prover - Prover Cer... | Systerel Smart Solv... | Nouvel onglet | SCADE Suite Design...

www.esterel-technologies.com/products/scade-suite/verification-validation/scade-suite-design-verifier/

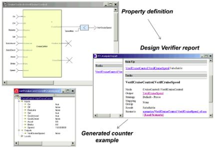
scade design verifier

Home » Products » SCADE Suite® » Verification and Validation » SCADE Suite Design Verifier

SCADE Suite Design Verifier

SCADE Suite Design Verifier (DV) is a formal verification assistant to formally express and assess safety requirements providing a productive way to find bugs early in the development process. Properties to verify are defined with SCADE itself. The boolean outputs are the proof objectives for DV that then automatically produces counter-examples. DV can also be used to find divisions by zero operations.

- Verification of safety properties expressed in SCADE Suite
- Automatic counter-example production in case of property failure
- Early detection of division-by-zero errors
- Easy and intuitive use of proof or bug-chasing modes



**SCADE Suite Design Verifier is based on Prover Plug-In®, a registered trademark of Prover Technology AB in Sweden, the United States and in other countries.*

prover plugged in

Verification and Validation

- SCADE Suite Simulator
- SCADE Suite Design Verifier
- SCADE Suite Timing and Stack Verifiers

Downloads

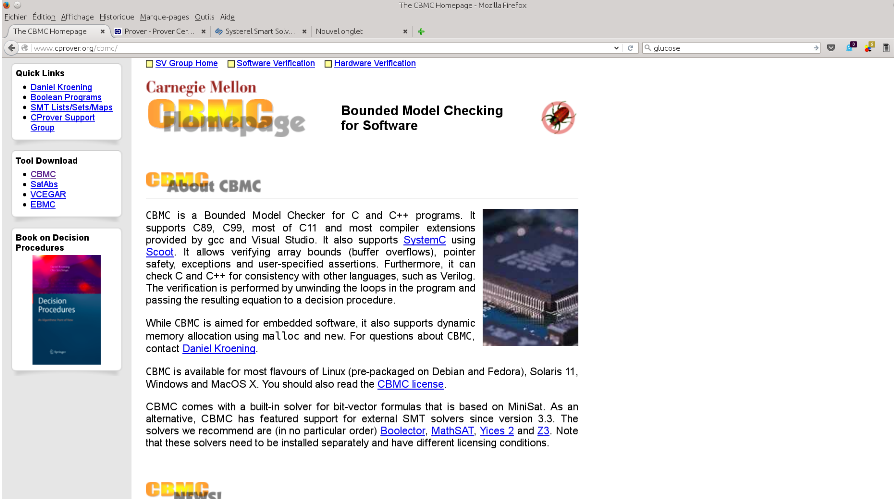
Handbook - DO-178C for Avionics
Datasheet - SCADE Suite Technical Datasheet
Training - Getting Started with SCADE Suite
Demos - SCADE Demos
White Paper - Understanding How SCADE Suite KCG Generates Safe C Code

Simulink Design Verifier

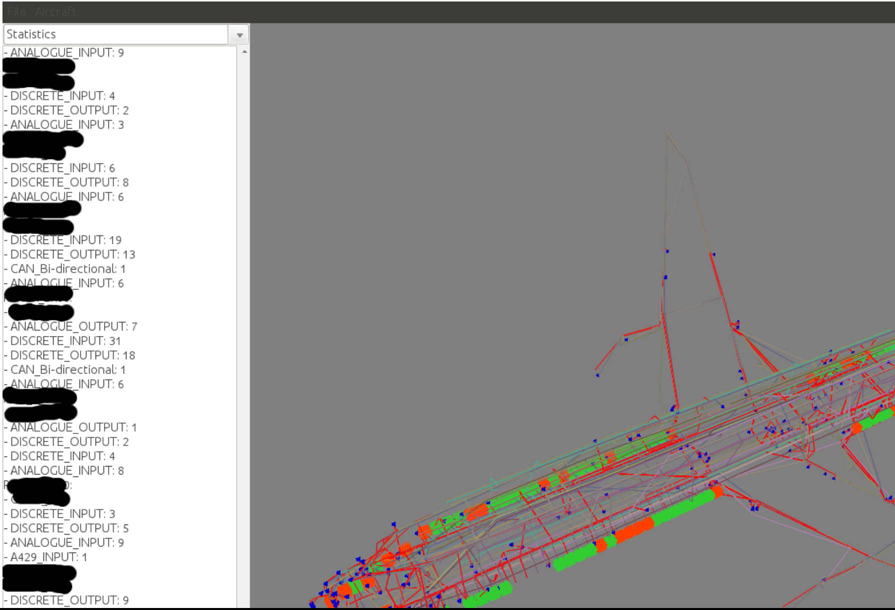
Systerel

Systerel Smart Solver, Vérification formelle de systèmes ou logiciels développés en SCADE, C, Ada - Mozilla FireFox
Fichier Édition Affichage Historique Marque-pages Outils Aide
The CBMC Homepage
Prover - Prover Cer...
Systerel Smart Solv...
Nouvel onglet
www.systerel.fr/innovation/produits/systerel-smart-solver/
glucose
Société Carrières Actualités Contacts
Français
Recherche
systerel
Solutions temps réel sécurisées
Solutions
Expertises
Innovation
Secteurs d'activité
Accueil › Innovation › Produits › Systerel Smart Solver
Systerel Smart Solver
systerel smart solver for C
systerel smart solver for Ada
systerel smart solver for SCADE
Vérification formelle de systèmes ou logiciels développés en SCADE, C, Ada
Systerel propose une gamme de solutions éprouvées industriellement pour l'analyse statique ou la preuve formelle de systèmes ou de logiciels à l'aide de technologies à base de Model Checking.
Systerel Smart Solver est un moteur d'analyse basé sur les technologies SAT.
Une large gamme de services
Associé à des traducteurs et outils spécialisés, Systerel Smart Solver offre une large gamme de services :
Analyse statique de code
Preuve de propriétés – certification
Preuve des propriétés définies par l'utilisateur avec outils d'analyse de contre-exemples
Génération automatique de tests (fonctionnels/structurels)

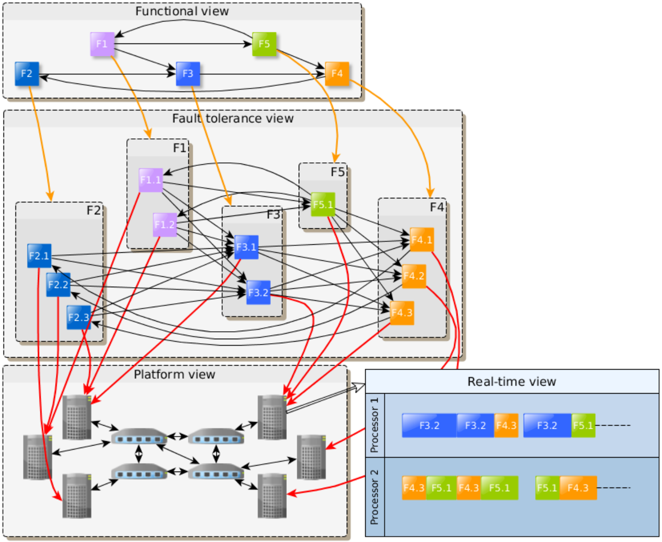
CProver


The CBMC Homepage - Mozilla Firefox
Fichier Édition Affichage Historique Marque-pages Outils Aide
The CBMC Homepage
Prover - Prover Cer...
Systerel Smart Solv...
Nouvel onglet
www.cprover.org/cbmc/
glucose
SV Group Home
Software Verification
Hardware Verification
Quick Links
Daniel Kroening
Boolean Programs
SMT Lists/Sets/Maps
CProver Support Group
Tool Download
CBMC
SatAbs
VCEGAR
EBMC
Book on Decision Procedures
Decision Procedures
Carnegie Mellon
CBMC Homepage
Bounded Model Checking for Software
About CBMC
CBMC is a Bounded Model Checker for C and C++ programs. It supports C89, C99, most of C11 and most compiler extensions provided by gcc and Visual Studio. It also supports SystemC using Scoot. It allows verifying array bounds (buffer overflows), pointer safety, exceptions and user-specified assertions. Furthermore, it can check C and C++ for consistency with other languages, such as Verilog. The verification is performed by unwinding the loops in the program and passing the resulting equation to a decision procedure.
While CBMC is aimed for embedded software, it also supports dynamic memory allocation using malloc and new. For questions about CBMC, contact Daniel Kroening.
CBMC is available for most flavours of Linux (pre-packaged on Debian and Fedora), Solaris 11, Windows and MacOS X. You should also read the CBMC license.
CBMC comes with a built-in solver for bit-vector formulas that is based on MiniSat. As an alternative, CBMC has featured support for external SMT solvers since version 3.3. The solvers we recommend are (in no particular order) Boolector, MathSAT, Yices 2 and Z3. Note that these solvers need to be installed separately and have different licensing conditions.

ONERA


File Aircraft
Statistics
- ANALOGUE_INPUT: 9
[redacted]
- DISCRETE_INPUT: 4
- DISCRETE_OUTPUT: 2
- ANALOGUE_INPUT: 3
[redacted]
- DISCRETE_INPUT: 6
- DISCRETE_OUTPUT: 8
- ANALOGUE_INPUT: 6
[redacted]
- DISCRETE_INPUT: 19
- DISCRETE_OUTPUT: 13
- CAN_Bi-directional: 1
- ANALOGUE_INPUT: 6
[redacted]
- ANALOGUE_OUTPUT: 7
- DISCRETE_INPUT: 31
- DISCRETE_OUTPUT: 18
- CAN_Bi-directional: 1
- ANALOGUE_INPUT: 6
[redacted]
- ANALOGUE_OUTPUT: 1
- DISCRETE_OUTPUT: 2
- DISCRETE_INPUT: 4
- ANALOGUE_INPUT: 8
[redacted]
- DISCRETE_INPUT: 3
- DISCRETE_OUTPUT: 5
- ANALOGUE_INPUT: 9
- A429_INPUT: 1
[redacted]
- DISCRETE_OUTPUT: 9

ONERA



Functional view
F1
F5
F2
F3
F4
Fault tolerance view
F1
F1.1
F1.2
F5
F5.1
F2
F2.1
F2.2
F2.3
F3
F3.1
F3.2
F4
F4.1
F4.2
F4.3
Platform view
Real-time view
Processor 1
F3.2
F3.2
F4.3
F3.2
F5.1
Processor 2
F4.3
F5.1
F4.3
F5.1
F5.1
F4.3

Outline

DPPL algorithm

```
if (propagate(F, V) == CONFLICT) { return UNSAT; }
dl = 0;
while( !satisfied(F, V) ) {
  (x, p)  = decide(F, V); // select variable and phase
  dl += 1;                // bump decision level
  V += (x -> p);          // extend assignment
  if (propagate(F, V) == CONFLICT) {
    (bl, ccl) = analyseConflict(F, V);
    if (bl < 0) { // backjump past root
      return UNSAT;
    } else { // backjump somewhere
      backjump(F, V, bl, ccl);
      dl = bl;
    }
  }
}
return SAT;
```

Two watched literals

How to detect **efficiently** that a clause is **unit**?

- associate to each variable x_i the set C_{x_i} of clauses in which x_i appears
- for each $c \in C_{x_i}$, **watch two literals** l and l'
- if x_i is assigned to T or F, for each $c \in C_{x_i}$
  - if l (resp. l') is X, the clause is **not unit**
  - if l (resp. l') is T or F, look for another literal l'' with value X in the clause
    - if such a literal exists, watch l'' instead of l (resp. l')
    - else, c is **unit**, propagate l' (resp. l).

Advantages: locality, nothing to modify when backjumping!

Clause learning

CDCL is performent mainly because it efficiently analyzes conflicts and learns clauses.

Clause learning

CDCL is performent mainly because it efficiently analyzes conflicts and learns clauses.

Some notations:

- **value fixing**
  At devision level d, fixing x_i to T is noted $x_i@d$, fixing x_i to F is noted $\bar{x}_i@d$
- **antecedents**
  When the value of x_i is fixed by **propagation** through a unit clause c, c is called the **antecedent** of x_i
- **implication graph**
  An implication graph is a graph in which
  - nodes represent variables fixings at decision levels
  - edges represent propagation and are labelled with antecedents
  - decisions are nodes without antecedent

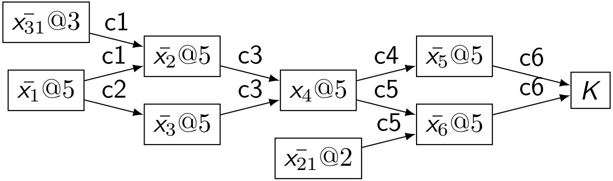
Example

$$F = \{c_1, c_2, c_3, c_4, c_5, c_6\}$$
$$c_1 = \{x_1, x_{31}, \bar{x_2}\}$$
$$c_2 = \{x_1, \bar{x_3}\}$$
$$c_3 = \{x_2, x_3, x_4\}$$
$$c_4 = \{\bar{x_4}, \bar{x_5}\}$$
$$c_5 = \{x_{21}, \bar{x_4}, \bar{x_6}\}$$
$$c_6 = \{x_5, x_6\}$$

let use suppose that the decisions are the following:

- $\bar{x_{21}}@2$
- $\bar{x_{31}}@3$
- $\bar{x_1}@5$ (current level)

`propagate` deduces at the same time x_5 and $\bar{x_5}$, or x_6 and $\bar{x_6}$, i.e. a **conflict** (K).



Conflict analysis

Starting from the conflict clause, a clause avoiding the conflict by Resolution is built by following the antecedents in the implication graph:

$$K \xrightarrow{\{x_5, x_6\}} \{x_5, x_6\} \xrightarrow{\{\bar{x_4}, \bar{x_5}\}} \{x_6, \bar{x_4}\} \xrightarrow{\{x_{21}, \bar{x_4}, \bar{x_6}\}} \{x_{21}, \bar{x_4}\} \xrightarrow{\{x_2, x_3, x_4\}}$$

$$\{x_2, x_3, x_{21}\} \xrightarrow{\{x_1, \bar{x_3}\}} \{x_2, x_{21}, x_1\} \xrightarrow{\{x_1, x_{31}, \bar{x_2}\}} \{x_1, x_{21}, x_{31}\}$$

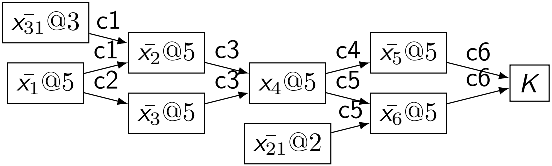
The conflict clause $\{x_{21}, x_{31}, x_1\}$ is added in the base of clauses and we backtrack to a decision level s.t. the clause is **unit** (to use a different phase for at least one variable in the clause).

Unique Implication Point

Reducing conflict clauses with Unique Implication Point (UIP):

- a UIP is a **dominant** of the conflict node in the propagation graph
- there is at least one UIP at each decision level (the decision itself)
- stop when the clause contains the first UIP of the current decision level and only decision variables from the lower levels

In the picture below, $x_4@5$ is the first UIP of level 5. The conflict clause is $\{x_{21}, \bar{x_4}\}$ instead of $\{x_{21}, x_{31}, x_1\}$.



Dynamic decision heuristics

Score each variable in conflict clause via Variable State Independent Decaying Sum (VSIDS), first seen in Chaff (2000)

- each variable is associated to an activity counter, initialized to 0
- the counter is incremented each time the variable appears in a conflict clause
- each *n* conflicts, divide the activity of each variable by a constant to concentrate on most recent conflicts

A revolutionary idea for 2000, improved in Berkmin, MiniSat, picosat, etc.

Phase heuristics

When backjumping, a part of the current solution is lost and this part could be coherent on a subset of the formula.

The “reasoning” must be redone several times.

Idea: when backjumping, save the current assignment of variables for **satisfied clauses** and use it as heuristics.

Allows to reuse lots of decisions.

And more!

- restarting
- data structures
- clauses erasing
- etc.

Conclusion

SAT solvers are used in real-life applications, particularly for formal methods:

- they can solve problems with billions of variables and clauses
- plenty of theories can be encoded in SAT:
  - arithmetics modulo 2^n
  - arithmetics with arbitrary precision
  - arrays
  - etc.
- lots of available solvers: zChaff, MiniSat, Glucose, SAT4J, Lingeling, …
- an annual competition to test solvers, SAT-COMP