

SAT and SMT Solvers in Practice

Marijn Heule and Ruben Martins

**Carnegie
Mellon
University**

<http://www.cs.cmu.edu/~mheule/15816-f23/>
<https://github.com/marijnheule/sat-examples.git>

Automated Reasoning and Satisfiability
September 11, 2023

DIMACS: SAT solver input format

The DIMACS format for SAT solvers has three types of lines:

- **header:** `p cnf n m` in which `n` denotes the highest variable index and `m` the number of clauses
- **clauses:** a sequence of integers ending with “0”
- **comments:** any line starting with “c ”

	c	example
	<code>p</code>	<code>cnf 4 7</code>
$(a \vee b \vee \bar{c}) \wedge$	<code>1</code>	<code>2 -3 0</code>
$(\bar{a} \vee \bar{b} \vee c) \wedge$	<code>-1</code>	<code>-2 3 0</code>
$(b \vee c \vee \bar{d}) \wedge$	<code>2</code>	<code>3 -4 0</code>
$(\bar{b} \vee \bar{c} \vee d) \wedge$	<code>-2</code>	<code>-3 4 0</code>
$(a \vee c \vee d) \wedge$	<code>1</code>	<code>3 4 0</code>
$(\bar{a} \vee \bar{c} \vee \bar{d}) \wedge$	<code>-1</code>	<code>-3 -4 0</code>
$(\bar{a} \vee b \vee d)$	<code>-1</code>	<code>2 4 0</code>

DIMACS: SAT solver output format

The solution line of a SAT solver starts with “s ”:

- s **SATISFIABLE**: The formula is satisfiable
- s **UNSATISFIABLE**: The formula is unsatisfiable
- s **UNKNOWN**: The solver cannot determine satisfiability

In case the formula is satisfiable, the solver emits a certificate:

- lines starting with “v ”
- a list of integers ending with 0
- e.g. v -1 2 4 0

In case the formula is unsatisfiable, then most solvers support emitting a **proof of unsatisfiability** to a separate file

CaDiCaL: download and install

Most SAT solvers are implemented in C/C++

CaDiCaL is one of the strongest SAT solvers. As the name suggests it is based on CDCL. Recommended for Linux and macOS users.

obtain CaDiCaL:

- `git clone https://github.com/arminbiere/cadical.git`
- `cd cadical`
- `./configure; make`

to run: `./build/cadical formula.cnf`

Kissat: download and install

Most SAT solvers are implemented in C/C++

Kissat is successor of CaDiCaL and it is written in C.
Recommended for Linux and macOS users.

obtain Kissat:

- `git clone`
`https://github.com/arminbiere/kissat.git`
- `cd kissat`
- `./configure; make`

to run: `./build/kissat formula.cnf`

SAT4J: download and install

SAT4J is a SAT solver in Java. It is also based on CDCL.
Recommended for windows users.

obtain SAT4J:

- `git clone`
`https://github.com/marijnheule/sat-examples.git`
- `cd sat-examples`

to run: `java -jar org.sat4j.core-2.3.1.jar formula.cnf`

UBCSAT: download and install

UBCSAT is a collection of local search SAT solvers.

obtain UBCSAT:

- download and unzip
`http://ubcsat.dtopkins.com/downloads/ubcsat-beta-12-b18.tar.gz`
- `cd ubcsat-beta-12-b18`
- `make clean; make`

to run: `./ubcsat -alg ddfw -i formula.cnf`

there are many LS algorithms to choose from (`-alg`)
`./ubcsat -ha` (shows the available algorithms)

YalSAT: download and install

YalSAT: **y**et **a**nother **l**ocal search **SAT** solver:

obtain YalSAT:

- `git clone`
`https://github.com/arminbiere/yalsat.git`
- `cd yalsat`
- `./configure.sh; make`

to run: `./yalsat formula.cnf`

A powerful local search solver from the author of CaDiCaL and Kissat

Many SAT solvers

Many SAT solvers have been developed

Lots of them participate in the annual SAT competition

- All code of participants in open source
- Each solver is run on hundreds of benchmarks
- Large timeout 5000 seconds

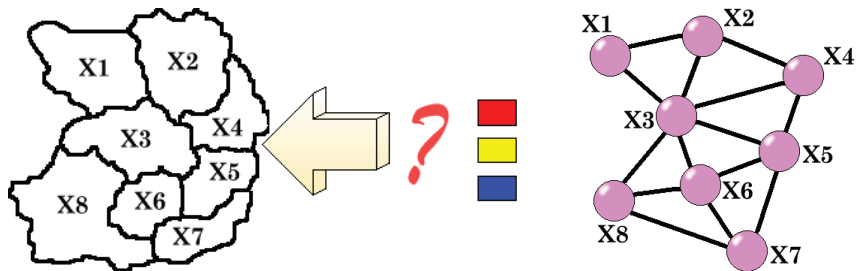
For details and downloading more solvers visit

<http://satcompetition.org/>

Demo: SAT Solving

Graph coloring

Given a graph $G(V, E)$, can the vertices be colored with k colors such that for each edge $(v, w) \in E$, the vertices v and w are colored differently.



Graph coloring encoding

Variables	Range	Meaning
$x_{v,i}$	$i \in \{1, \dots, c\}$ $v \in \{1, \dots, V \}$	node v has color i
Clauses	Range	Meaning
$(x_{v,1} \vee x_{v,2} \vee \dots \vee x_{v,c})$	$v \in \{1, \dots, V \}$	v is colored
$(\bar{x}_{v,s} \vee \bar{x}_{v,t})$	$s \in \{1, \dots, c-1\}$ $t \in \{s+1, \dots, c\}$	v has at most one color
$(\bar{x}_{v,i} \vee \bar{x}_{w,i})$	$(v, w) \in E$	v and w have a different color

Graph coloring encoding code

```
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char** argv) {
    FILE* graph = fopen (argv[1], "r");
    int i, j, a, b, nVertex, nEdge, nColor = atoi (argv[2]);
    fscanf (graph, " p edge %i %i ", &nVertex, &nEdge);

    printf ("p cnf %i %i\n", nVertex * nColor, nVertex + nEdge * nColor);

    for (i = 0; i < nVertex; i++) {
        for (j = 1; j <= nColor; j++)
            printf ("%i ", i * nColor + j);
        printf ("\n"); }

    while (1) {
        int tmp = fscanf (graph, " e %i %i ", &a, &b);
        if (tmp == 0 || tmp == EOF) break;
        for (j = 1; j <= nColor; j++)
            printf ("-%i -%i 0\n", (a-1) * nColor + j, (b-1) * nColor + j);
    }
}
```

Demo: Encode, Decode

Unsatisfiable cores

An **unsatisfiable core** of an unsatisfiable formula F is a subset of F that is unsatisfiable.

An **minimal unsatisfiable core** of an unsatisfiable formula such that the removal of any clause makes the formula satisfiable.

Extracting a minimal unsatisfiable core from a formula has **many applications**, but the computational costs could be high.

- maxSAT
- diagnosis
- formal verification

Proofs

A **proof of unsatisfiability** is a certificate that a given formula is unsatisfiable.

Various proof producing methods exists (another lecture).

Proof checking tools cannot only validate a proof but also produce **additional information** about the formula:

- unsatisfiable core
- optimized proof

DRAT-trim is a tool that validates proofs and produces such information

Demo: Core Extraction



StarExec is a cross community logic solving service

- Great to evaluate solvers/heuristics in parallel
- Also used to run the SAT/SMT competitions

Register at <https://www.starexec.org/>

- select SAT as your community

Demo: StarExec

Tools for making SAT-based modeling easier

PySAT is a Python toolkit that makes it easier for users to call SAT solvers and build encodings using Python:

- <https://pysathq.github.io/>
- SAT solver is still written in C, C++
- Interface includes several encodings for linear constraints:
 - At-most-one constraints
 - Cardinality constraints
 - AIGER circuits to CNF
 - ...
- Well documented
- Active development

Demo: PySAT

SMT-LIB: SMT solver input format (I)

<http://smtlib.cs.uiowa.edu/>

Language has similarities with functional languages and it is more readable than CNF. Theories:

- Arrays,
- Bitvectors,
- Boolean predicates,
- Floating point,
- Ints,
- Reals

SMT-LIB: SMT solver input format (II)

```
; Basic Boolean example
(set-logic QF_UF)
(declare-const p Bool)
(assert (and p (not p)))
(check-sat) ; returns 'unsat'
(exit)
```

SMT-LIB: SMT solver input format (III)

```
; Integer arithmetic
(set-logic QF_LIA)
(declare-const x Int)
(declare-const y Int)
(assert (= (- x y) (+ x (- y) 1)))
(check-sat) ; returns 'unsat'
(exit)
```


SMT Solvers

- Z3 (Microsoft):
<https://github.com/Z3Prover/z3/wiki>
- CVC5 (Stanford): <https://cvc5.github.io/>
- Yices (SRI): <http://yices.csl.sri.com/>
- Boolector (JKU Austria):
<https://boolector.github.io/>

SMT Solvers

We recommend the use of **Z3**:

- Tutorial:
`https://theory.stanford.edu/~nikolaj/programmingz3.html`
- APIs for Python, C++, Java
- MIT License: `https://github.com/Z3Prover/z3`
- Most used and cited SMT solver (>9,500 citations)

Proving program equivalence in SMT

```
1 int power3(int in)
2 {
3     int i, out_a;
4     out_a = in;
5     for (i = 0; i < 2; i++)
6         out_a = out_a * in;
7     return out_a;
8 }
```

```
1 int power3_new(int in)
2 {
3     int out_b;
4
5     out_b = (in * in) * in;
6
7     return out_b;
8 }
```

$$\varphi_a \equiv (\text{out0_a} = \text{in0_a}) \wedge (\text{out1_a} = \text{out0_a} \times \text{in0_a}) \wedge \\ (\text{out2_a} = \text{out1_a} \times \text{in0_a})$$

$$\varphi_b \equiv \text{out0_b} = (\text{in0_b} \times \text{in0_b}) \times \text{in0_b}$$

To show these programs are equivalent, we must show the following formula is valid: $\text{in0_a} = \text{in0_b} \wedge \varphi_a \wedge \varphi_b \implies \text{out2_a} = \text{out0_b}$

Demo: Program equivalence with SMT solving (BV)

```
1 (declare-fun out0_a () (_ BitVec 128))
2 (declare-fun out1_a () (_ BitVec 128))
3 (declare-fun in0_a () (_ BitVec 128))
4 (declare-fun out2_a () (_ BitVec 128))
5 (declare-fun out0_b () (_ BitVec 128))
6 (declare-fun in0_b () (_ BitVec 128))
7 (define-fun phi_a () Bool
8     (and (= out0_a in0_a) ; out0_a = in0_a
9         (and (= out1_a (bvmul out0_a in0_a)) ; out1_a = out0_a * in0_a
10            (= out2_a (bvmul out1_a in0_a))))); out2_a = out1_a * in0_a
11 (define-fun phi_b () Bool
12     (= out0_b (bvmul (bvmul in0_b in0_b) in0_b))); out0_b = in0_b * in0_b * in0_b
13 (define-fun phi_input () Bool
14     (= in0_a in0_b))
15 (define-fun phi_output () Bool
16     (= out2_a out0_b ))
17 (assert (not (=> (and phi_input phi_a phi_b) phi_output )))
18 (check-sat)
```

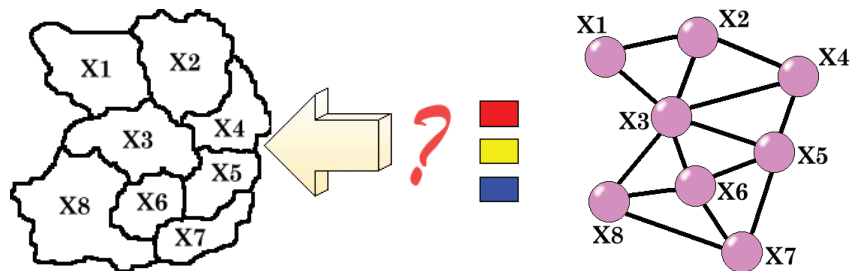
Demo: Program equivalence with SMT solving (Int)

```
1 (declare-fun out0_a () (Int))
2 (declare-fun out1_a () (Int))
3 (declare-fun in0_a () (Int))
4 (declare-fun out2_a () (Int))
5 (declare-fun out0_b () (Int))
6 (declare-fun in0_b () (Int))
7 (define-fun phi_a () Bool
8     (and (= out0_a in0_a) ; out0_a = in0_a
9         (and (= out1_a (* out0_a in0_a)) ; out1_a = out0_a * in0_a
10            (= out2_a (* out1_a in0_a))))); out2_a = out1_a * in0_a
11 (define-fun phi_b () Bool
12     (= out0_b (* (* in0_b in0_b) in0_b))); out0_b = in0_b * in0_b * in0_b
13 (define-fun phi_input () Bool
14     (= in0_a in0_b))
15 (define-fun phi_output () Bool
16     (= out2_a out0_b ))
17 (assert (not (=> (and phi_input phi_a phi_b) phi_output )))
18 (check-sat)
```

Demo: Program equivalence with SMT solving (UF)

```
1 (declare-fun out0_a () (_ BitVec 128))
2 (declare-fun out1_a () (_ BitVec 128))
3 (declare-fun in0_a () (_ BitVec 128))
4 (declare-fun out2_a () (_ BitVec 128))
5 (declare-fun out0_b () (_ BitVec 128))
6 (declare-fun in0_b () (_ BitVec 128))
7 (declare-fun f ((_ BitVec 128) (_ BitVec 128)) (_ BitVec 128))
8 (define-fun phi_a () Bool
9     (and (= out0_a in0_a) ; out0_a = in0_a
10         (and (= out1_a (f out0_a in0_a)) ; out1_a = out0_a * in0_a
11             (= out2_a (f out1_a in0_a)))) ; out2_a = out1_a * in0_a
12 (define-fun phi_b () Bool
13     (= out0_b (f (f in0_b in0_b) in0_b))) ; out0_b = in0_b * in0_b * in0_b
14 (define-fun phi_input () Bool
15     (= in0_a in0_b))
16 (define-fun phi_output () Bool
17     (= out2_a out0_b ))
18 (assert (not (=> (and phi_input phi_a phi_b) phi_output )))
19 (check-sat)
```

Graph coloring encoding in SMT



Variables:

- Integer variables x_i for each node

Constraints:

- $1 \leq x_i \leq c$
- $x_i \neq x_j$ for $(x_i, x_j) \in E$

Graph coloring encoding code

```
from z3 import *
import sys

with open(sys.argv[1]) as f:
    content = f.readlines()

nodes=int(content[0].split()[2])
edges=int(content[0].split()[3])
s = Solver()
variables = []

for id in range(1,nodes+1):
    variables.append(Int('x'+str(id)))
    # each node can be assigned a value between 1 and k
    s.add(And(1 <= variables[id-1], variables[id-1] <= int(sys.argv[2])))

for line in content:
    if line[0]=='p':
        continue
    else:
        edge=line.split()
        # if two nodes are connected then they have different colors
        s.add((variables[int(edge[1])-1])!=(variables[int(edge[2])-1]))

s.check() # check if sat/unsat
print(s.model()) # print model
```


Demo: Encoding in SMT

Unsatisfiable cores in SMT

```
; Getting unsatisfiable cores
(set-option :produce-unsat-cores true)
(set-logic QF_UF)
(declare-const p Bool) (declare-const q Bool) (declare-const r Bool)
(declare-const s Bool) (declare-const t Bool)
(assert (! ( $\Rightarrow$  p q) :named PQ))
(assert (! ( $\Rightarrow$  q r) :named QR))
(assert (! ( $\Rightarrow$  r s) :named RS))
(assert (! ( $\Rightarrow$  s t) :named ST))
(assert (! (not ( $\Rightarrow$  q s)) :named NQS))
(check-sat)
; unsat
(get-unsat-core)
; (QR RS NQS)
(exit)
```