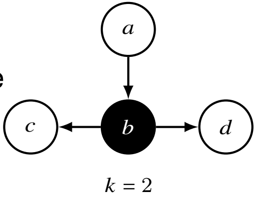
PROPOSITIONAL LOGIC: APPLICATIONS OF SAT SOLVING

Course “Computational Logic”

Wolfgang Schreiner
Research Institute for Symbolic Computation (RISC)
Wolfgang.Schreiner@risc.jku.at

JOHANNES KEPLER UNIVERSITY LINZ

Example: The Graph Coloring Problem



Given a graph and k colors, assign to every graph node some color such that all neighbor nodes have different colors.

- Propositional variables: $a_1, a_2, b_1, b_2, c_1, c_2, d_1, d_2$.
  - Variable v_i: graph node v has color i.
- Every node has some color.

$$a_1 \lor a_2, b_1 \lor b_2, c_1 \lor c_2, d_1 \lor d_2.$$

- No node has multiple colors.

$$\neg(a_1 \land a_2), \neg(b_1 \land b_2), \neg(c_1 \land c_2), \neg(d_1 \land d_2).$$

- All neighbor nodes have different colors.

$$\neg(a_1 \land b_1), \neg(a_2 \land b_2),$$
$$\neg(c_1 \land b_1), \neg(c_2 \land b_2),$$
$$\neg(d_1 \land b_1), \neg(d_2 \land b_2).$$

Are these constraints satisfiable?

The Graph Coloring Problem with MiniSAT

```
debian10!1> cat colors.cnf
  c file "colors.cnf"
  p cnf 8 14
  1 2 0
  3 4 0
  5 6 0
  7 8 0
  -1 -2 0
  -3 -4 0
  -5 -6 0
  -7 -8 0
  -1 -3 0
  -2 -4 0
  -3 -5 0
  -4 -6 0
  -3 -7 0
  -4 -8 0
```

```
debian10!2> minisat -verb=0 colors.cnf colors.out
WARNING: for repeatability, setting FPU to use double precision
SATISFIABLE
debian10!3> cat colors.out
SAT
-1 2 3 -4 -5 6 -7 8 0
```

The Graph Coloring Problem with Limboole

Limboole on the Go!

Uses Limboole (MIT licensed), PicoSAT (MIT licensed), and DepQBF (GPLv3 licensed) to parse an easy SAT and QBF DSL (instead of relying on DIMACS). Compiled using Emscripten, Source Code and Modifications are available on GitHub. Created by Max Heisinger. I also wrote a short blog entry about this. Support on GitHub and on #limboole on Libera.Chat.

Open How-To

Satisfiability Check

Run (or Shift+Enter in input area)

Input Drag&Drop ✓

```
(a1 | a2) & (b1 | b2) & (c1 | c2) & (d1 | d2) &
!(a1 & a2) & !(b1 & b2) & !(c1 & c2) & !(d1 & d2) &
!(a1 & b1) & !(a2 & b2) &
!(c1 & b1) & !(c2 & b2) &
!(d1 & b1) & !(d2 & b2)
```

Output

```
% SATISFIABLE formula (satisfying assignment follows)
a1 = 1
a2 = 0
b1 = 0
b2 = 1
c1 = 1
c2 = 0
d1 = 1
d2 = 0
```

Errors

Example: Sudoku

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

Fill an $N \times N$ grid with numbers $1..N$ such that the same number does not occur twice in any row or column or any of the N subgrids of size $\sqrt{N} \times \sqrt{N}$.

- N^3 variables $c_{i,j,v}$
  - Variable $c_{i,j,v}$: cell in row i and column j has value v.
- Every field has some value.
  - N^2 formulas $c_{i,j,1} \vee \ldots \vee c_{i,j,N}$.
- No field has two different values.
  - $N^3 \cdot (N+1)/2$ formulas $\neg(c_{i,j,v_1} \wedge c_{i,j,v_2})$.
- Every row has not in two different columns the same value.
  - $N^3 \cdot (N+1)/2$ formulas $\neg(c_{i,j_1,v} \wedge c_{i,j_2,v})$.
- Every column has not in two different rows the same value.
  - $N^3 \cdot (N+1)/2$ formulas $\neg(c_{i_1,j,v} \wedge c_{i_2,j,v})$.
- Every subgrid has in two different cells not the same value.
  - $N^3 \cdot (N+1)/2$ formulas $\neg(c_{i',j',v} \wedge c_{i'',j'',v})$.

We better write a program to generate the formulas.

A Java Program for Printing the Formula

```
public class Sudoku {
  private static final int M=2; private static final int N=M*M;

  public static void main(String[] args) {
    someNumber(); and(); notMultipleNumbers(); and();
    rowConditions(); and(); columnConditions(); and(); tileConditions();
  }

  private static void or() { System.out.print("|"); }
  private static void and() { System.out.print("&"); }
  private static void not() { System.out.print("!"); }
  private static void open() { System.out.print("("); }
  private static void close() { System.out.print(")"); }
  private static void entry(int i, int j, int v)
  { System.out.print("cell_" + i + "_" + j + "_value_" + (v+1)); }
  private static void entries(int i1, int j1, int v1, int i2, int j2, int v2)
  { open(); entry(i1,j1,v1); and(); entry(i2,j2,v2); close(); }
  ...
}
```

A Java Program for Printing the Formula

```
private static void someNumber() {
  for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
      open();
      for (int v = 0; v < N; v++) { entry(i,j,v); if (v+1 < N) or(); }
      close();
      if (i+1 < N || j+1 < N) and();
}}}

private static void notMultipleNumbers() {
  for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
      for (int v1 = 0; v1 < N-1; v1++) {
        for (int v2 = v1+1; v2 < N; v2++) {
          not(); entries(i,j,v1,i,j,v2);
          if (i+1 < N || j+1 < N || v1+1 < N-1 || v2+1 < N) and();
}}}}}
```

A Java Program for Printing the Formula

```
private static void rowConditions() {
  for (int i = 0; i < N; i++) {
    for (int j1 = 0; j1 < N-1; j1++) {
      for (int j2 = j1+1; j2 < N; j2++) {
        for (int v = 0; v < N; v++)  {
          not(); entries(i,j1,v,i,j2,v);
          if (i+1 < N || j1+1 < N-1 || j2+1 < N || v+1 < N) and();
}}}}}

private static void columnConditions() {
  for (int i = 0; i < N; i++) {
    for (int j1 = 0; j1 < N-1; j1++) {
      for (int j2 = j1+1; j2 < N; j2++) {
        for (int v = 0; v < N; v++) {
          not(); entries(j1,i,v,j2,i,v);
          if (i+1 < N || j1+1 < N-1 || j2+1 < N || v+1 < N) and();
}}}}}
```

A Java Program for Printing the Formula

```
private static void tileConditions() {
  for (int i = 0; i < M; i++) {
    for (int j = 0; j < M; j++) {
      for (int v = 0; v < N; v++) {
        for (int k1 = 0; k1 < N-1; k1++) {
          for (int k2 = k1+1; k2 < N; k2++) {
            int k1i = k1/M; int k1j = k1%M;
            int k2i = k2/M; int k2j = k2%M;
            not(); entries(i*M+k1i,j*M+k1j,v,i*M+k2i,j*M+k2j,v);
            if (i+1 < M || j+1 < M || v+1 < N || k1+1 < N-1 || k2+1 < N) and();
}}}}}}
```

A Java Program for Printing the Formula

```
debian10!1> java Sudoku > sudoku2.bool
debian10!2> cat sudoko2.bool
(cell_0_0_value_1|cell_0_0_value_2|cell_0_0_value_3|cell_0_0_value_4)&
(cell_0_1_value_1|cell_0_1_value_2|cell_0_1_value_3|cell_0_1_value_4)&
...&
!(cell_0_0_value_1&cell_0_0_value_2)&
!(cell_0_0_value_1&cell_0_0_value_3)&
...&
!(cell_0_0_value_1&cell_0_1_value_1)&
!(cell_0_0_value_2&cell_0_1_value_2)&
...&
!(cell_0_0_value_1&cell_1_0_value_1)&
!(cell_0_0_value_2&cell_1_0_value_2)
...&
!(cell_0_0_value_1&cell_0_1_value_1)&
!(cell_0_0_value_1&cell_1_0_value_1)
...
```

Size of formula is 15 KB for $N = 2$, 444 KB for $N = 3$.

Sudoku with Limboole

Limboole on the Go!

Uses Limboole (MIT licensed), PicoSAT (MIT licensed), and DepQBF (GPLv3 licensed) to parse an easy SAT and QBF DSL (instead of relying on DIMACS). Compiled using Emscripten, Source Code and Modifications are available on GitHub. Created by Max Heisinger. I also wrote a short blog entry about this. Support on GitHub and on #limboole on Libera.Chat.

Open How-To

Satisfiability Check

Run (or Shift+Enter in input area)

Input Drag&Drop ✓

```
(cell_0_0_value_1|cell_0_0_value_2|cell_0_0_value_3|cell_0_0_value_4)&
(cell_0_1_value_1|cell_0_1_value_2|cell_0_1_value_3|cell_0_1_value_4)&
(cell_0_2_value_1|cell_0_2_value_2|cell_0_2_value_3|cell_0_2_value_4)&
(cell_0_3_value_1|cell_0_3_value_2|cell_0_3_value_3|cell_0_3_value_4)&
(cell_1_0_value_1|cell_1_0_value_2|cell_1_0_value_3|cell_1_0_value_4)&
(cell_1_1_value_1|cell_1_1_value_2|cell_1_1_value_3|cell_1_1_value_4)&
(cell_1_2_value_1|cell_1_2_value_2|cell_1_2_value_3|cell_1_2_value_4)&
(cell_1_3_value_1|cell_1_3_value_2|cell_1_3_value_3|cell_1_3_value_4)&
(cell_2_0_value_1|cell_2_0_value_2|cell_2_0_value_3|cell_2_0_value_4)&
(cell_2_1_value_1|cell_2_1_value_2|cell_2_1_value_3|cell_2_1_value_4)&
(cell_2_2_value_1|cell_2_2_value_2|cell_2_2_value_3|cell_2_2_value_4)&
(cell_2_3_value_1|cell_2_3_value_2|cell_2_3_value_3|cell_2_3_value_4)&
(cell_3_0_value_1|cell_3_0_value_2|cell_3_0_value_3|cell_3_0_value_4)&
(cell_3_1_value_1|cell_3_1_value_2|cell_3_1_value_3|cell_3_1_value_4)&
(cell_3_2_value_1|cell_3_2_value_2|cell_3_2_value_3|cell_3_2_value_4)&
(cell_3_3_value_1|cell_3_3_value_2|cell_3_3_value_3|cell_3_3_value_4)
&!(cell_0_0_value_1&cell_0_0_value_2)&!
(cell_0_0_value_1&cell_0_0_value_3)&!
```

Output

```
% SATISFIABLE formula (satisfying assignment follows)
cell_0_0_value_1 = 1
cell_0_0_value_2 = 0
cell_0_0_value_3 = 0
cell_0_0_value_4 = 0
cell_0_1_value_1 = 0
cell_0_1_value_2 = 1
cell_0_1_value_3 = 0
cell_0_1_value_4 = 0
cell_0_2_value_1 = 0
cell_0_2_value_2 = 0
cell_0_2_value_3 = 1
cell_0_2_value_4 = 0
cell_0_3_value_1 = 0
cell_0_3_value_2 = 0
cell_0_3_value_3 = 0
cell_0_3_value_4 = 1
cell_1_0_value_1 = 0
```

Sudoku with Limboole

```
% SATISFIABLE formula (...)
  cell_0_0_value_1 = 1
  cell_0_1_value_2 = 1
  cell_0_2_value_3 = 1
  cell_0_3_value_4 = 1
  cell_1_0_value_3 = 1
  cell_1_1_value_4 = 1
  cell_1_2_value_1 = 1
  cell_1_3_value_2 = 1
  cell_2_0_value_2 = 1
  cell_2_1_value_1 = 1
  cell_2_2_value_4 = 1
  cell_2_3_value_3 = 1
  cell_3_0_value_4 = 1
  cell_3_1_value_3 = 1
  cell_3_2_value_2 = 1
  cell_3_3_value_1 = 1
```

1	2	3	4
3	4	1	2
2	1	4	3
4	3	2	1

Sudoku with Limboole: Preset Cell Entries

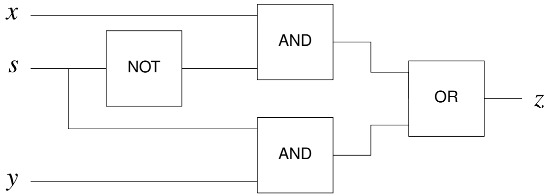
```
cell_0_0_value_3 & cell_0_2_value_4 & cell_2_3_value_1 & cell_3_1_value_2 & ... (printed formula)
```

```
% SATISFIABLE formula (...)
  cell_0_0_value_3 = 1
  cell_0_2_value_4 = 1
  cell_2_3_value_1 = 1
  cell_3_1_value_2 = 1
  cell_0_1_value_1 = 1
  cell_0_3_value_2 = 1
  cell_1_0_value_2 = 1
  cell_1_1_value_4 = 1
  cell_1_2_value_1 = 1
  cell_1_3_value_3 = 1
  cell_2_0_value_4 = 1
  cell_2_1_value_3 = 1
  cell_2_2_value_2 = 1
  cell_3_0_value_1 = 1
  cell_3_2_value_3 = 1
  cell_3_3_value_4 = 1
```

3	1	4	2
2	4	1	3
4	3	2	1
1	2	3	4

Example: A Multiplexer

- We can model digital circuits as propositional formulas.
  - Wires: propositional variables.
  - Gates: logical connectives.
- A one bit multiplexer:




```
z <-> (x & !s) | (y & s)
```

Thus digital circuits can be analyzed by SAT solvers.

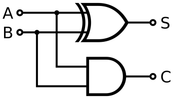
Example: A Multiplexer

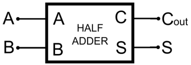
```
debian10!1> limboole
(z <-> (x & !s) | (y & s)) -> (s -> (z <-> y))
% VALID formula
debian10!2> limboole
(z <-> (x & !s) | (y & s)) -> (!s -> (z <-> x))
% VALID formula
debian10!3> limboole
(z <-> (x & !s) | (y & s)) -> ((z <-> y) -> s)
% INVALID formula (falsifying assignment follows)
z = 0
x = 0
s = 0
y = 0
debian10!4> limboole
(z <-> (x & !s) | (y & s)) -> ((z <-> y) -> s | (x <-> y))
% VALID formula
```

The multiplexer shows the expected behavior.

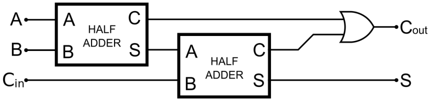
Example: An Arithmetic Circuit

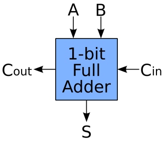
Let us now consider digital arithmetic.



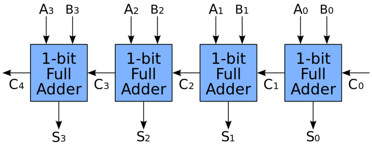


A	B	S	C_{out}
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1





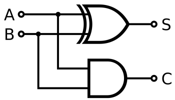
A	B	C_{in}	S	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

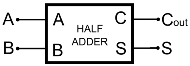


	C_4	A_3 B_3	A_2 B_2	A_1 B_1	A_0 B_0	C_0
		1	0	1	1	
+		0	1	1	0	1
=	1	1	1	1	0	

From a "half adder" to a "full adder" to a "ripple-carry adder".

A Half Adder in OCaml



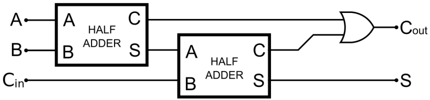


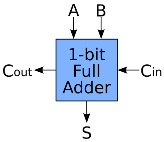
A	B	S	C_{out}
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

```
let halfsum x y = Iff(x,Not y);;
let halfcarry x y = And(x,y);;
let ha x y s c = And(Iff(s,halfsum x y),Iff(c,halfcarry x y));;
```

The (sub)circuits are modeled by formulas that are parameterized over other formulas; the parameters can be instantiated by atoms whose truth values determine the bit values of the wires.

A Full Adder in OCaml





A	B	C_{in}	S	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

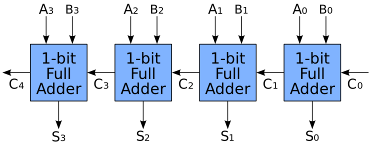
```
let fa0 x y z s c s0 c0 c1 = And(And(ha x y s0 c0,ha s0 z s c1),Iff(c,Or(c0,c1)));;

let carry x y z = Or(And(x,y),And(Or(x,y),z));;
let sum x y z = halfsum (halfsum x y) z;;
let fa x y z s c = And(Iff(s,sum x y z),Iff(c,carry x y z));;

let p(x) = Atom(P(x));;
let adder1 = fa0 (p "A") (p "B") (p "C_in") (p "S") (p "C_out") (p "S0") (p "C0") (p "C1");;
let adder2 = fa  (p "A") (p "B") (p "C_in") (p "S") (p "C_out");;

# tautology(Imp(adder1,adder2));;
- : bool = true
```

A Ripple-Carry Adder in OCaml



	C_4	A_3 B_3	A_2 B_2	A_1 B_1	A_0 B_0	C_0
		1	0	1	1	
+		0	1	1	0	1
=	1	1	1	1	0	

```
let conjoin f l = list_conj (map f l);;
let ripplecarry x y c out n =
  conjoin (fun i -> fa (x i) (y i) (c i) (out i) (c(i + 1))) (0 -- (n - 1));;

let mk_index x i = Atom(P(x^"_"^(string_of_int i)));;
let [x; y; out; c] = map mk_index ["A"; "B"; "S"; "C"];;

# ripplecarry x y c out 2;;
<<((S_0 <=> (A_0 <=> ~B_0) <=> ~C_0) /\ (C_1 <=> A_0 /\ B_0 \/ (A_0 \/ B_0) /\ C_0)) /\
   (S_1 <=> (A_1 <=> ~B_1) <=> ~C_1) /\ (C_2 <=> A_1 /\ B_1 \/ (A_1 \/ B_1) /\ C_1)>>
```

Analyzing the Adder

Does the adder ensure associativity?

```
let [x; y; z; xy; yz; xyz1; xyz2; cxy; cyz; cxyz1; cxyz2] =
  map mk_index ["X"; "Y"; "Z";
                "XY";  "YZ";  "XYZ1";  "XYZ2";
                "CXY"; "CYZ"; "CXYZ1"; "CXYZ2"];;

let bvsize = ...;;

let addxy   = ripplecarry x  y  cxy   xy   bvsize;;
let addxyz1 = ripplecarry xy z  cxyz1 xyz1 bvsize;;
let adder1 = And(addxy,addxyz1);;

let addyz   = ripplecarry y  z  cyz   yz   bvsize;;
let addxyz2 = ripplecarry x  yz cxyz2 xyz2 bvsize;;
let adder2 = And(addyz,addxyz2);;
```

Two circuits that compute $(x + y) + z$ respectively $x + (y + z)$.

Analyzing the Adder

Does the adder ensure associativity?

```
let pvar x i = p(x^"_"^(string_of_int i));;
let rec allequal x y i =
  if i = 1 then Iff(pvar x 0, pvar y 0)
           else And(Iff(pvar x (i-1), pvar y (i-1)), allequal x y (i-1));;

let associative =
  Imp(And(And(adder1, adder2), And(
      And(Not(p "CXY_0"),Not(p "CXYZ1_0")),
      And(Not(p "CYZ_0"),Not(p "CXYZ2_0")))),
    (allequal "XYZ1" "XYZ2" bvsize));;
```

If both circuits initially provide carry 0 as input for their components, they two bit vectors produced by the circuits are identical in all positions.

Analyzing the Adder

For bitvector size 2, the OCaml implementation of DPLL with CDCL is adequate.

```
# let associative = ... ;;
<<(((((XY_0 <=> (X_0 <=> ~Y_0) <=> ~CXY_0) /\ (CXY_1 <=> X_0 /\ Y_0 \/ (X_0 \/ Y_0) /\ CXY_0)) /\
 (XY_1 <=> (X_1 <=> ~Y_1) <=> ~CXY_1) /\ (CXY_2 <=> X_1 /\ Y_1 \/ (X_1 \/ Y_1) /\ CXY_1)) /\
((XYZ1_0 <=> (XY_0 <=> ~Z_0) <=> ~CXYZ1_0) /\ (CXYZ1_1 <=> XY_0 /\ Z_0 \/ (XY_0 \/ Z_0) /\ CXYZ1_0)) /\
(XYZ1_1 <=> (XY_1 <=> ~Z_1) <=> ~CXYZ1_1) /\ (CXYZ1_2 <=> XY_1 /\ Z_1 \/ (XY_1 \/ Z_1) /\ CXYZ1_1)) /\
 (((YZ_0 <=> (Y_0 <=> ~Z_0) <=> ~CYZ_0) /\ (CYZ_1 <=> Y_0 /\ Z_0 \/ (Y_0 \/ Z_0) /\ CYZ_0)) /\
(YZ_1 <=> (Y_1 <=> ~Z_1) <=> ~CYZ_1) /\ (CYZ_2 <=> Y_1 /\ Z_1 \/ (Y_1 \/ Z_1) /\ CYZ_1)) /\
 ((XYZ2_0 <=> (X_0 <=> ~YZ_0) <=> ~CXYZ2_0) /\ (CXYZ2_1 <=> X_0 /\ YZ_0 \/ (X_0 \/ YZ_0) /\ CXYZ2_0)) /\
 (XYZ2_1 <=> (X_1 <=> ~YZ_1) <=> ~CXYZ2_1) /\ (CXYZ2_2 <=> X_1 /\ YZ_1 \/ (X_1 \/ YZ_1) /\ CXYZ2_1)) /\
(~CXY_0 /\ ~CXYZ1_0) /\ ~CYZ_0 /\ ~CXYZ2_0 ==> (XYZ1_1 <=> XYZ2_1) /\ (XYZ1_0 <=> XYZ2_0)>>
```

```
# dplbtaut associative;;
   - : bool = true
```

For larger bitvector sizes, we need to employ a more efficient SAT solver.

Analyzing the Adder

A translation of formulas to Limboole syntax.

```
let rec limboole f =
  match f with
    Atom(P(x)) -> x | Not(f0) -> "(!" ^ (limboole f0) ^ ")"
  | And(f1,f2) -> "(" ^ (limboole f1) ^ "&" ^ (limboole f2) ^ ")"
  | Or(f1,f2) ->  "(" ^ (limboole f1) ^ "|" ^ (limboole f2) ^ ")"
  | Imp(f1,f2) -> "(" ^ (limboole f1) ^ "->" ^ (limboole f2) ^ ")"
  | Iff(f1,f2) -> "(" ^ (limboole f1) ^ "<->" ^ (limboole f2) ^ ")"
  | _ -> failwith "unsupported formula"
;;
print_string (limboole associative);;
```

```
alan!318> limboole
(((((((XY_0<->((X_0<->(!Y_0))<->(!CXY_0)))&(CXY_1<->((X_0&Y_0)|((X_0|Y_0)&CXY_0))))&((XY_1<->((X_1<->(!Y_1))<->(!CXY_1)))&
(CXY_2<->((X_1&Y_1)|((X_1|Y_1)&CXY_1)))))&(((XYZ1_0<->((XY_0<->(!Z_0))<->(!CXYZ1_0)))&(CXYZ1_1<->((XY_0&Z_0)|((XY_0|Z_0)&
CXYZ1_0))))&((XYZ1_1<->((XY_1<->(!Z_1))<->(!CXYZ1_1)))&(CXYZ1_2<->((XY_1&Z_1)|((XY_1|Z_1)&CXYZ1_1))))))&
((((YZ_0<->((Y_0<->(!Z_0))<->(!CYZ_0)))&(CYZ_1<->((Y_0&Z_0)|((Y_0|Z_0)&CYZ_0))))&((YZ_1<->((Y_1<->(!Z_1))<->(!CYZ_1)))&
(CYZ_2<->((Y_1&Z_1)|((Y_1|Z_1)&CYZ_1)))))&(((XYZ2_0<->((X_0<->(!YZ_0))<->(!CXYZ2_0)))&(CXYZ2_1<->((X_0&YZ_0)|((X_0|YZ_0)&
CXYZ2_0))))&((XYZ2_1<->((X_1<->(!YZ_1))<->(!CXYZ2_1)))&(CXYZ2_2<->((X_1&YZ_1)|((X_1|YZ_1)&CXYZ2_1)))))))&
(((!CXY_0)&(!CXYZ1_0))&((!CYZ_0)&(!CXYZ2_0))))->((XYZ1_1<->XYZ2_1)&(XYZ1_0<->XYZ2_0)))
% VALID formula
```

Now the problem can be easily solved also for bit vector size 32.