An Introduction to SMT Solving

Shaowei Cai

Institute of Software, Chinese Academy of Sciences

2022.4.17

Many examples inspired/borrowed from Andrew Reynolds’s and Nikolaj Bjørner’s slides

Outline

- SMT Basis
- Lazy Approach --- DPLL(T)
- Eager Approach --- Bit Blasting

The Logic Languages

SAT: Propositional Satisfiability

$(\text{Tie} \lor \text{Shirt}) \land (\neg \text{Tie} \lor \neg \text{Shirt}) \land (\neg \text{Tie} \lor \text{Shirt})$

FOL: First-order Logic

$\forall X,Y,Z\ [X*Y*Z] = (X*Y)*Z]$

$\forall X[X*inv(X)=e]\ \forall X[X*e=e]$

$\forall n \in \{z | z > 2, z \in Z\} \neg \exists x, y, z \in \mathrm{Z}\ (x^n + y^n = z^n)$

SMT: Satisfiability Modulo background Theories

$b+2 = c \land A[3] \neq A[c-b+1]$

First Order Logic (FOL)

- First-order logic (FOL), also called predicate logic and the first-order predicate calculus.
- FOL extends propositional logic with predicates, functions, and quantifiers.

  - variables x, y, z, x1, x2, . . .
  - constants a, b, c, a1, a2, ...
  - Terms evaluate to values other than truth values, integers, people, or cards of a deck. //objects
    - More complicated terms are constructed using functions.

Example: these are terms
a, a constant (or 0-ary function);
x, a variable;
f(a), a unary function f applied to a constant;
g(x, b), a binary function g applied to a variable x and a constant b;
f(g(x, f(b))).

First Order Logic (FOL)

- First-order logic (FOL), also called predicate logic and the first-order predicate calculus.
- FOL extends propositional logic with predicates, functions, and quantifiers.

  - Predicates P, Q… // properties, relations of objects
    - An n-ary predicate takes n terms as arguments.
    - Example: x is a student S(x)
      - Andy is a student S(Andy)
      - Bob is not a student $\neg S(Bob)$
    - Example: y is a teacher of x T(y, x)
      - John is a teacher of Andy T(John, Andy)
  - An atom is ⊤, ⊥, or an n-ary predicate applied to n terms.

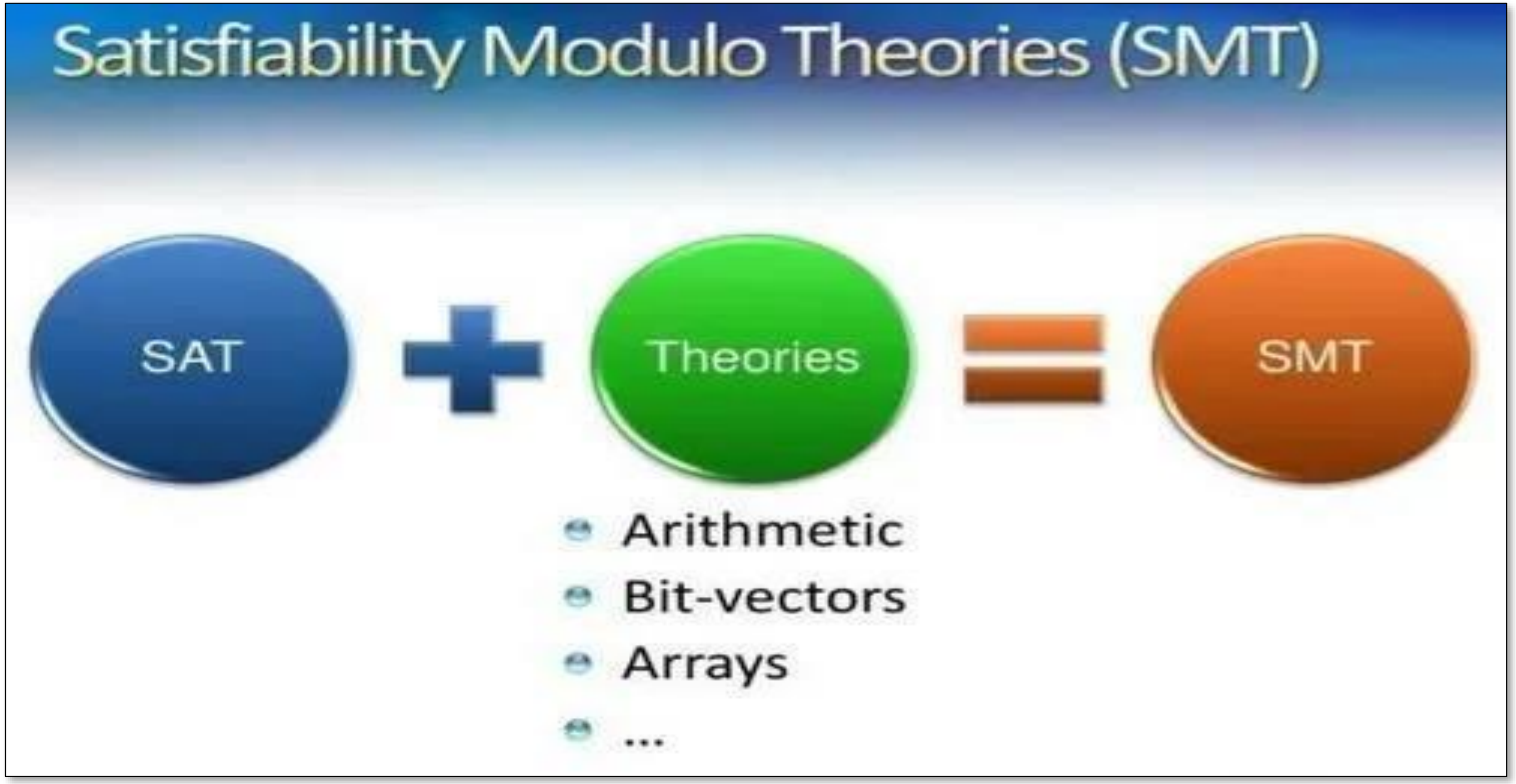
  - A literal is an atom or its negation.

- First-order logic (FOL), also called predicate logic and the first-order predicate calculus.
- FOL extends propositional logic with predicates, functions, and quantifiers.

  - Quantifiers
  - the existential quantifier ∃x. F[x], read "there exists an x such that F[x]";
  - the universal quantifier ∀x. F[x], read "for all x, F[x]".

  - A FOL formula is
    - a literal,
    - the application of a logical connective ¬, ∧, ∨, →, or ↔ to a formula or formulae,
    - or the application of a quantifier to a formula.

Satisfiability Modulo Theories



From Propositional to Quantifier-Free Theories

Example:
$\phi := (x_1 - x_2 \leq 13 \vee x_2 \neq x_3) \wedge (x_2 = x_3 \rightarrow x_4 > x_5) \wedge A \wedge \neg B$

Propositional Skeleton $\mathrm{PS}_{\Phi} = (b_1 \vee \neg b_2) \wedge (b_2 \rightarrow b_3) \wedge A \wedge \neg B$

$b_1 : x_1 - x_2 \leq 13$

$b_2 : x_2 = x_3$

$b_3 : x_4 > x_5$

From Propositional to Quantifier-Free Theories

Example:

- $\mathrm{a} = \mathrm{b} + 2 \wedge A = write(B, a + 1,4) \wedge (read(A, b + 3) = 2 \vee f(a - 1) \neq f(b + 1))$

- Propositional Skeleton $\mathrm{PS}_\Phi = y_1 \wedge y_2 \wedge (y_3 \vee y_4)$

- y_1: $\mathrm{a} = \mathrm{b} + 2$
- y_2: $A = write(B, a + 1,4)$
- y_3: $read(A, b + 3) = 2$
- y_4: $f(a - 1) \neq f(b + 1)$

Language: Signatures

- A first-order theory T is defined by the following components.

  1. Its signature Σ is a set of constant, function, and predicate symbols.
     - A constant can also be viewed as a 0-ary function
     - A FOL propositional variable is a 0-ary predicate, which we write A, B, C, …
  2. Its set of axioms $\mathcal{A}$ is a set of closed FOL formulae in which only constant, function, and predicate symbols of Σ appear.

  - A Σ-formula is constructed from constant, function, and predicate symbols of Σ, as well as variables, logical connectives, and quantifiers.

  - As usual, the symbols of Σ are just symbols without prior meaning.
  - The axioms A provide their meaning.

Interpretation

Recall

- An interpretation I assigns to every propositional variable exactly one truth value. For example, I : {P $\longmapsto$ $true$, Q $\longmapsto$ false, ...}

- A formula F is satisfiable iff there exists an interpretation I such that I ⊨ F.

- A formula F is valid iff for all interpretations I, I ⊨ F

Interpretation

- FOL interpretation I: (D_I, α_I)
- The domain D_I of an interpretation I is a nonempty set of values or objects, such as integers, real numbers, dogs, people, or merely abstract objects…

The **assignment** α_I of interpretation I maps constant, function, and predicate symbols to elements, functions, and predicates over D_I. It also maps variables to elements of D_I:

- each variable symbol x is assigned a value x_I from D_I;
- each n-ary function symbol f is assigned an n-ary function

$$f_I : D_I^n \to D_I$$

  that maps n elements of D_I to an element of D_I;
- each n-ary predicate symbol p is assigned an n-ary predicate

$$p_I : D_I^n \to \{\text{true}, \text{ false}\}$$

  that maps n elements of D_I to a truth value.

Interpretation

Example

- F : x + y > z → y > z − x

- We construct a “standard” interpretation I
- The domain is the integers, $\mathbb{Z}$: $D_I = \mathbb{Z} = \{\dots, -2, -1, 0, 1, 2, \dots\}$
- α_I: $\{+ \mapsto +_{\mathbb{Z}}, - \mapsto -_{\mathbb{Z}}, > \mapsto >_{\mathbb{Z}}, x \mapsto 13, y \mapsto 42, x \mapsto 1\}$

T-satisfiability

- Given a FOL formula F and interpretation $I: (D_I, \alpha_I)$, we want to compute if F evaluates to true under interpretation I, I ⊨ F, or if F evaluates to false under interpretation I, I ⊭F.
  - I satisfies F: I ⊨ F

- T – interpretation: an interpretation satisfying I ⊨ A for every A $\in \mathcal{A}$.

- A Σ-formula F is satisfiable in T , or T -satisfiable, if there is a T-interpretation I that satisfies F.

Input Format : SMT-LIB2

- First, directives. E.g., asking models to be reported:

(set - option : produce - models true)

- Second, set background theory:

(set - logic QF_LIA)

- Standard theories of interest :
    - QF_BV: quantifier-free bit vector theory
    - QF_LRA : quantifier-free linear real arithmetic
    - QF_LIA: quantifier-free linear integer arithmetic
    - QF_NRA : quantifier-free nonlinear real arithmetic
    - QF_NIA : quantifier-free nonlinear integer arithmetic
    - …

Input Format : SMT-LIB2

- Third, declare variables

(declare-fun x () s), or (declare-const x s) //introducing new symbols `x` of sort `s`

common sorts: `Int Bool Real (_BitVec 3) ((_FixedSizedList 4) Real) (Set (_BitVec 3))`

E.g., integer variable x:

(declare - fun x () Int)

E.g., real variable z_1_3:

(declare - fun z_1_3 () Real)

Input Format : SMT-LIB2

- Fourth, assert formula.
- Expressions should be written in prefix form:

  (< operator > < arg 1 > … < arg n >)

```
( assert
  ( and
    ( or
      ( <=  (+  x  3)  ( *  2  u ) )
      ( >=  (+  v  4)   y )
      ( >=  (+  x  y  z )   2)
    )
    ( = 7
      (+
        ( ite  ( and  ( <=  x  2 )  ( <= 2  (+  x  3  (- 1) ) ) )  3  0)
        ( ite  ( and  ( <=  u  2 )  ( <= 2  (+  u  3  (- 1) ) ) )  4  0)
      )
    )
  )
)
```

- and, or, + have arbitrary arity
- \- is unary or binary
- * is binary
- ite is the if-then-else operator (like ? in C, C++, Java).

Let a be Boolean and b, c have the same sort S, then (ite a b c) is the expression of sort S equal to:

- b if a holds
- c if a does not hold

Input Format : SMT-LIB2

- Finally ask the SMT solver to check satisfiability …

(check - sat)

- … and report the model

(get - model)

- Anything following a ; up to an end-of-line is a comment
  - You can also use (set-info : comments) to write comments in your files

Input Format : SMT-LIB2

```
( set - option  :  produce - models true )
( set - logic   QF_LIA )
( declare - fun x ()    Int )
( declare - fun y ()    Int )
( declare - fun z ()    Int )          ;          This  is  an  example
( declare - fun u ()    Int )
( declare - fun v ()    Int )
( assert
  ( and
     ( or
       ( <= (+  x  3) (*  2  y ) )
       ( >= (+  x  4)   z )
     )
  (<= x  y))
)
( check - sat )
( get - model )
```

Input Format : SMT-LIB2

```
(set-logic QF_LIA)
(declare-fun x ()   Int)
(declare-fun y ()   Int)
(declare-fun z ()   Int)
(assert ( or  ( >  x  y)  ( >  x  z ) ) )
(assert ( or  ( <  ( +  x  1 )  y )  ( not  ( >  x  y ) ) ) )
(assert ( or  ( >  x  y )  ( >  z  y ) ) ) )
(check-sat)
```

Example

Input Format : SMT-LIB2

```
; There is a fast way to check that fixed size numbers are powers of two.
; It turns out that a bit-vector x is a power of two or zero if and only if x & (x - 1) is zero,
where & represents the bitwise and.
; When using Z3, if you do not set logic, it means all logics supported in Z3.

(define-fun is-power-of-two ((x (_ BitVec 4))) Bool
  (= #x0 (bvand x (bvsub x #x1))))
(declare-const a (_ BitVec 4))
(assert
 (not (= (is-power-of-two a)
         (or (= a #x0)
             (= a #x1)
             (= a #x2)
             (= a #x4)
             (= a #x8)))))
(check-sat)
```

Output Format : SMT-LIB2

- 1st line is sat or unsat
- If satisfiable, then comes a description of the solution in a model expression, where the value of each variable is given by:

(define – fun < variable > () < sort > < value >)

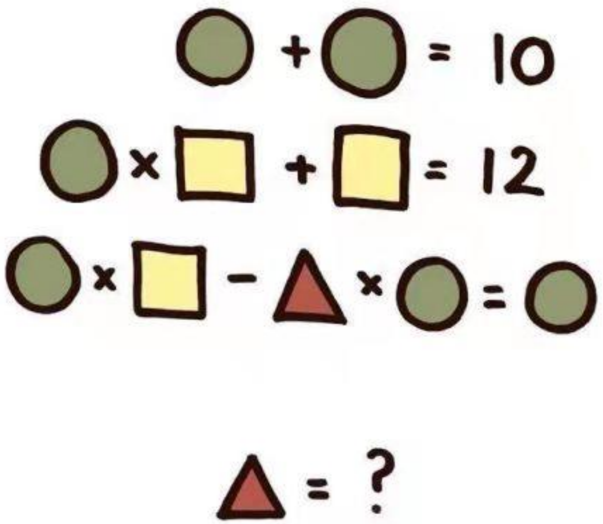
- Example:

```
sat
( model
    ( define - fun y ()   Int 0 )
    ( define - fun x ()   Int (- 3) )
    ( define - fun z ()   Int 2 )
)
```

SMT Encoding (Programming) – solving equations

It's that easy to solve it in Z3:

```
#!/usr/bin/python
from z3 import *

circle, square, triangle = Ints('circle square triangle')
s = Solver()
s.add(circle+circle==10)
s.add(circle*square+square==12)
s.add(circle*square-triangle*circle==circle)
print s.check()
print s.model()
```




```
sat
[triangle = 1, square = 2, circle = 5]
```

SMT Encoding (Programming) - Sudoku

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

```
% time python sudoku2_Z3.py
1 4 5 3 2 7 6 9 8
8 3 9 6 5 4 1 2 7
6 7 2 9 1 8 5 4 3
4 9 6 1 8 5 3 7 2
2 1 8 4 7 3 9 5 6
7 5 3 2 9 6 4 8 1
3 6 7 5 4 2 8 1 9
9 8 4 7 6 1 2 3 5
5 2 1 8 3 9 7 6 4

real     0m0.382s
user     0m0.346s
sys      0m0.036s
```

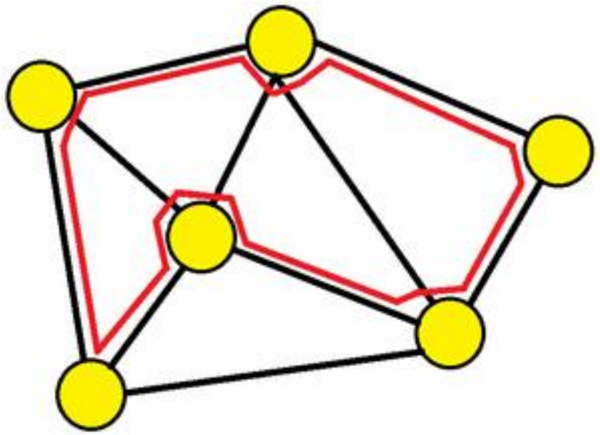
SMT-solvers are so helpful, in that our Sudoku solver has nothing else, we have just defined relationships between variables (cells).

```python
#!/usr/bin/env python3
import sys
from z3 import *
"""
------------------------------
00 01 02 | 03 04 05 | 06 07 08
10 11 12 | 13 14 15 | 16 17 18
20 21 22 | 23 24 25 | 26 27 28
------------------------------
30 31 32 | 33 34 35 | 36 37 38
40 41 42 | 43 44 45 | 46 47 48
50 51 52 | 53 54 55 | 56 57 58
------------------------------
60 61 62 | 63 64 65 | 66 67 68
70 71 72 | 73 74 75 | 76 77 78
80 81 82 | 83 84 85 | 86 87 88
------------------------------
"""
#https://sat-smt.codes/current_tree/puzzles/sudoku/1/sudoku2_Z3.py

s=Solver()
# using Python list comprehension , construct array of arrays of BitVec instances:
cells=[[Int('cell%d%d' % (r, c)) for c in range(9)] for r in range(9)]
puzzle="
..53.....8......2..7..1.5..4....53...1..7...6..32...8..6.5....9..4....3......97.."
```

```python
# process text line:
current_column=0
current_row=0
for i in puzzle:
    if i!='.':
        s.add(cells[current_row][current_column]==int(i))
        current_column=current_column+1
    if current_column==9:
        current_column=0
        current_row=current_row+1

for r in range(9):
    for c in range(9):
        s.add(cells[r][c]>=1)
        s.add(cells[r][c]<=9)

# for all 9 rows
for r in range(9):
    s.add(Distinct(cells[r][0],
    cells[r][1],
    cells[r][2],
    cells[r][3],
    cells[r][4],
    cells[r][5],
    cells[r][6],
    cells[r][7],
    cells[r][8]))
# for all 9 columns
for c in range(9):
s.add(Distinct(cells[0][c],
cells[1][c],
cells[2][c],
cells[3][c],
cells[4][c],
cells[5][c],
cells[6][c],
cells[7][c],
cells[8][c]))
```

```python
# enumerate all 9 squares
for r in range(0, 9, 3): # with each step of 3
    for c in range(0, 9, 3):
        # add constraints for each 3*3 square:
        s.add(Distinct(cells[r+0][c+0],
        cells[r+0][c+1],
        cells[r+0][c+2],
        cells[r+1][c+0],
        cells[r+1][c+1],
        cells[r+1][c+2],
        cells[r+2][c+0],
        cells[r+2][c+1],
        cells[r+2][c+2]))

print (s.check())
#print (s.model())
m=s.model()
for r in range(9):
    for c in range(9):
        sys.stdout.write (str(m[cells[r][c]])+" ")
print ("")
```

SMT Encoding (Programming) – Hamiltonian cycle



A **Hamiltonian path** (or traceable path) is a path in an undirected or directed graph that visits each vertex exactly once
A **Hamiltonian cycle** (or Hamiltonian circuit) is a Hamiltonian path that is a cycle.
NP complete problem.

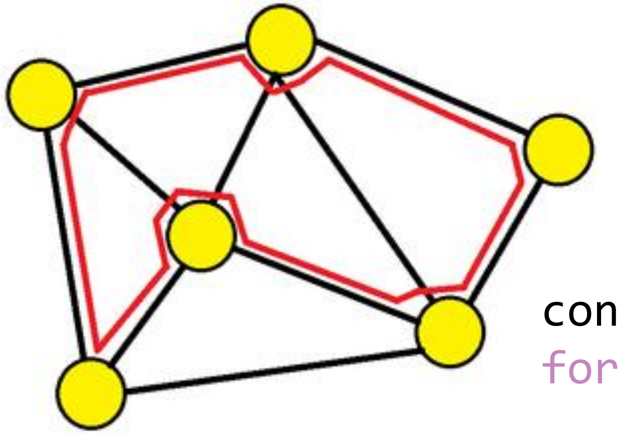
The position of every node in hamiltonian cycle order array should be a integer in [0, N).

$$\forall\, i \in [0, 1, \dots, N - 1](pos[i] \in [0, 1, \dots, N - 1] \wedge pos[i] \in \mathrm{Z})$$

For every node, there should be one node which is just next to it in hamiltonian cycle order.

$$\forall\, i \in [0, 1, \dots, N - 1] \exists j \{j \in [0, 1, \dots, N - 1] \wedge edge(i, j) \in G \wedge pos[j] \equiv (pos[i] + 1) \% N\}$$

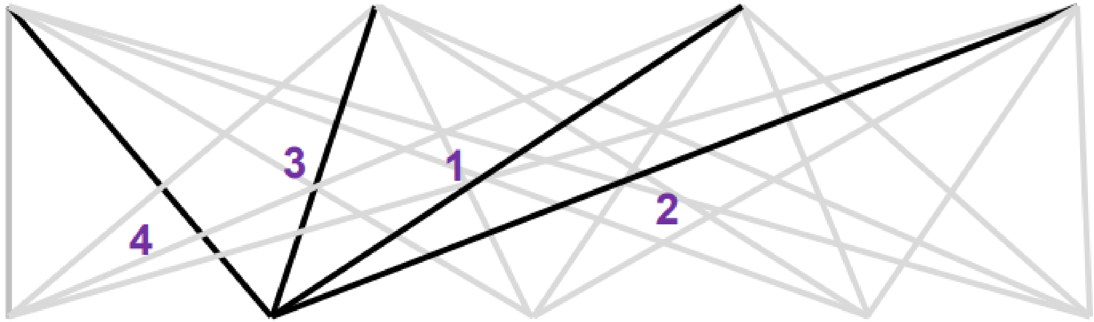
SMT Encoding (Programming) – Hamiltonian cycle



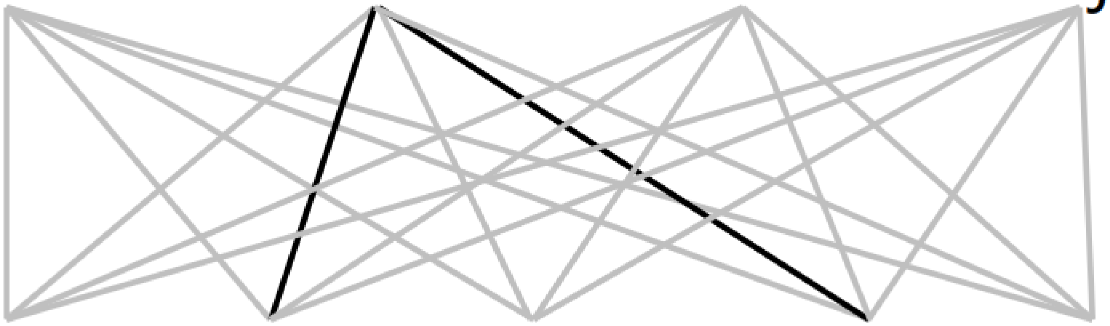
```
constraint <- {}
for i : {i | i in [0, N)} do
    constraint.add_clause(0 <= pos[i] < N and is_integer(pos[i]))
end for
constraint.add_clause(pos[0] == 0)
for i : {i | i in [0, N)} do
    or_clause <- {}
    for j : {j | node j can be reached by node i in graph} do
        or_clause.add_literal(pos[j] == (pos[i] + 1) % N)
    end for
    constraint.add_clause(or_clause)
end for
```

SMT Encoding (Programming) – Job Scheduling

Precedence: between two tasks of the same job



Resource: Machines execute at most one job at a time

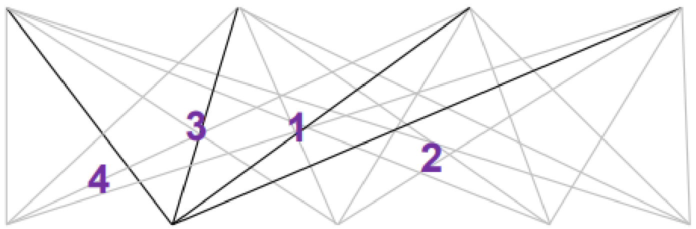


$$[start_{2,2}..end_{2,2}] \cap [start_{4,2}..end_{4,2}] = \emptyset$$

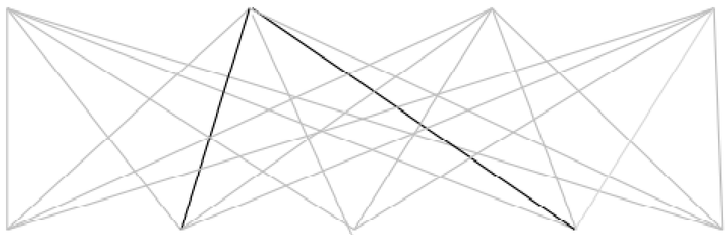
SMT Encoding (Programming) – Job Scheduling

Constraints:

Precedence:



Resource:



$[start_{2,2}..end_{2,2}] \cap [start_{4,2}..end_{4,2}] = \emptyset$

Encoding:

$t_{2,3}$ - start time of job 2 on mach 3

$d_{2,3}$ - duration of job 2 on mach 3

$t_{2,3} + d_{2,3} \leq t_{2,4}$

Not convex

$t_{2,2} + d_{2,2} \leq t_{4,2}$
$\vee$
$t_{4,2} + d_{4,2} \leq t_{2,2}$

SMT Encoding (Programming) – Job Scheduling

$d_{i,j}$	Machine 1	Machine 2
Job 1	2	1
Job 2	3	1
Job 3	2	3

$max = 8$

Solution

$t_{1,1} = 5,\ t_{1,2} = 7,\ t_{2,1} = 2,$
$t_{2,2} = 6,\ t_{3,1} = 0,\ t_{3,2} = 3$

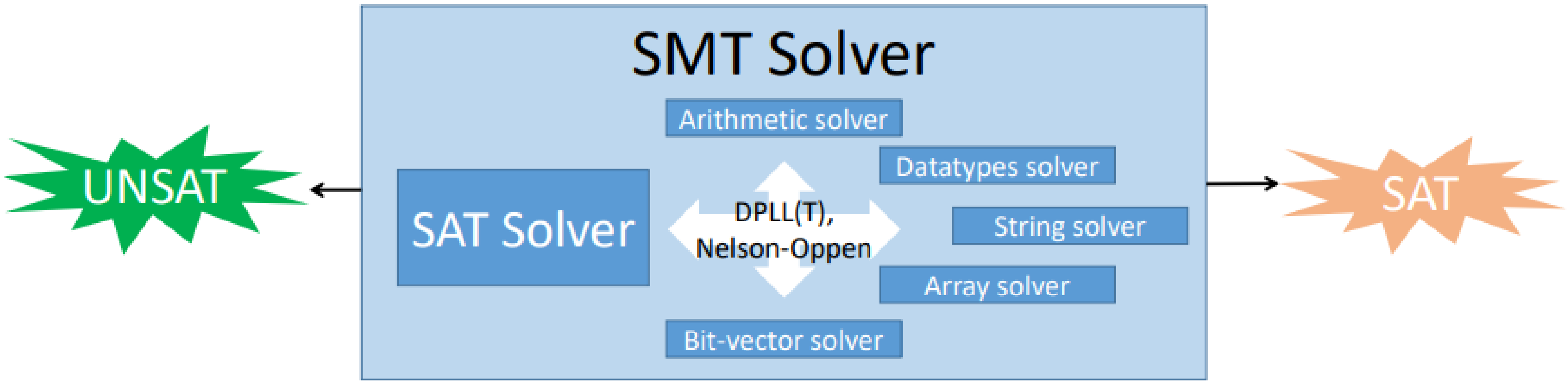
Encoding

$$
\begin{aligned}
&(t_{1,1} \geq 0) \wedge (t_{1,2} \geq t_{1,1} + 2) \wedge (t_{1,2} + 1 \leq 8) \wedge \\
&(t_{2,1} \geq 0) \wedge (t_{2,2} \geq t_{2,1} + 3) \wedge (t_{2,2} + 1 \leq 8) \wedge \\
&(t_{3,1} \geq 0) \wedge (t_{3,2} \geq t_{3,1} + 2) \wedge (t_{3,2} + 3 \leq 8) \wedge \\
&((t_{1,1} \geq t_{2,1} + 3) \vee (t_{2,1} \geq t_{1,1} + 2)) \wedge \\
&((t_{1,1} \geq t_{3,1} + 2) \vee (t_{3,1} \geq t_{1,1} + 2)) \wedge \\
&((t_{2,1} \geq t_{3,1} + 2) \vee (t_{3,1} \geq t_{2,1} + 3)) \wedge \\
&((t_{1,2} \geq t_{2,2} + 1) \vee (t_{2,2} \geq t_{1,2} + 1)) \wedge \\
&((t_{1,2} \geq t_{3,2} + 3) \vee (t_{3,2} \geq t_{1,2} + 1)) \wedge \\
&((t_{2,2} \geq t_{3,2} + 3) \vee (t_{3,2} \geq t_{2,2} + 1))
\end{aligned}
$$

Outline

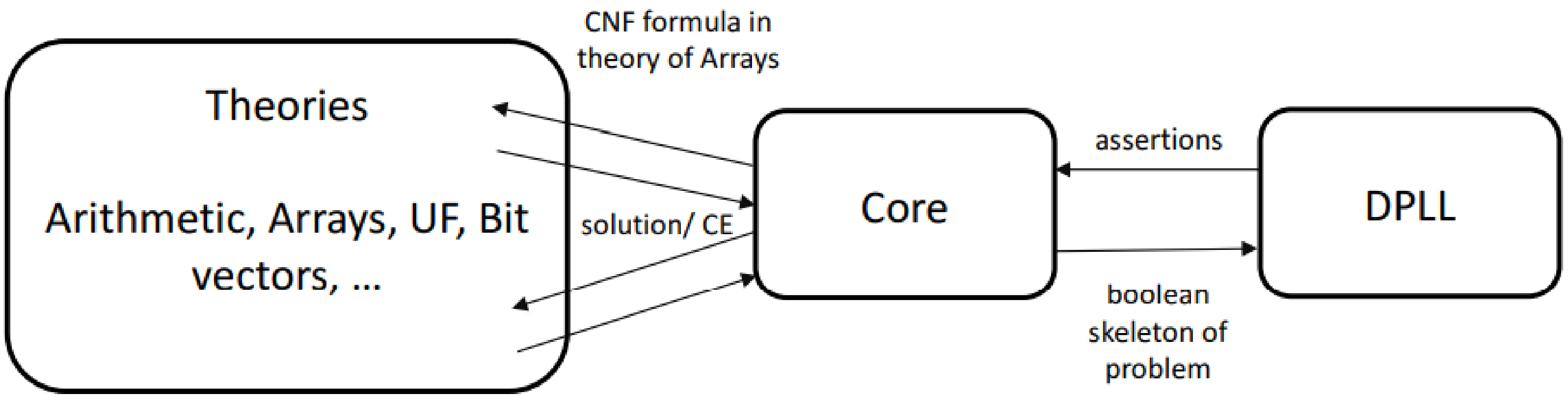
- SMT Basis
- Lazy Approach --- DPLL(T)
- Eager Approach --- Bit Blasting

DPLL(T)



UNSAT
SMT Solver
Arithmetic solver
Datatypes solver
SAT Solver
DPLL(T),
Nelson-Oppen
String solver
Array solver
Bit-vector solver
SAT

DPLL(T)



DPLL(T):

- “a general method—a framework, really—that generalizes CDCL to a decision procedure for decidable quantifier-free first order theories.
- The method is commonly referred to as DPLL(T), emphasizing that it is parameterized by a theory T.
- The fact that it is called DPLL(T) and not CDCL(T) is attributed to historical reasons only: it is based on modern CDCL solvers”

---”Decision Procedures” Daniel Kroening, Ofer Strichman

Before the search

<u>Abstract the skeleton</u>:

Given atom a, we associate with it a unique Boolean variable e(a), which we call the Boolean encoder of this atom.

$$\varphi := \; x = y \vee x = z$$

$$e(\varphi) := \; e(x = y) \vee e(x = z)$$

<u>Congruence Closure</u>

a=b, f(a,f(b,g(a)))=d, g(b)=c, f(a,c)=c, f(a,c)≠d

a=b

f(a,f(b,g(a)))=d

g(b)=c=f(a,c)=g(a)

f(b,g(a))

…merge congruent terms

since a=b and c=g(a), we know f(a,c)=f(b,g(a))

Propositional Skeleton

$$\varphi := \; x = y \wedge ((y = z \wedge \neg(x = z)) \vee x = z)\ .$$

The propositional skeleton of φ is

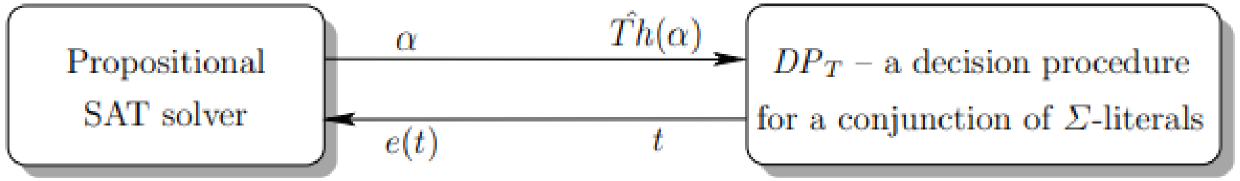
$$e(\varphi) := \; e(x = y) \wedge ((e(y = z) \wedge \neg e(x = z)) \vee e(x = z))\ .$$

Let $\mathcal{B}$ be a Boolean formula, initially set to $e(\varphi)$, i.e.,

$$\mathcal{B} := \; e(\varphi)\ .$$

$$\alpha := \; \{e(x = y) \mapsto \text{TRUE},\; e(y = z) \mapsto \text{TRUE},\; e(x = z) \mapsto \text{FALSE}\}\ .$$

A basic lazy approach



$$\varphi := \ x = y \wedge ((y = z \wedge \neg(x = z)) \vee x = z)\ .$$

The propositional skeleton of φ is

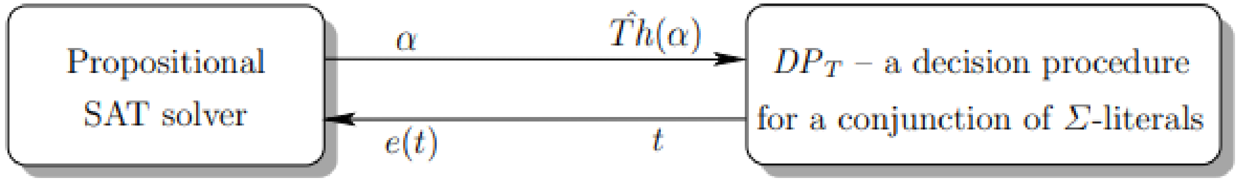
$$e(\varphi) := \ e(x = y) \wedge ((e(y = z) \wedge \neg e(x = z)) \vee e(x = z))\ .$$

Let $\mathcal{B}$ be a Boolean formula, initially set to $e(\varphi)$, i.e.,

$$\mathcal{B} := \ e(\varphi)\ .$$

- Call SAT solver to solve $e(\varphi)$, find $\alpha := \ \{e(x = y) \mapsto \text{TRUE},\ e(y = z) \mapsto \text{TRUE},\ e(x = z) \mapsto \text{FALSE}\}\ .$
- →Call decision procedure DP_T to check the conjunction corresponding to α, denoted by $\widehat{Th}(\alpha)$, $\widehat{Th}(\alpha) := x{=}y \wedge y = z \wedge \neg(x{=}z)$ → the result: $\widehat{Th}(\alpha)$ is unsat.
- → B is conjoined with $e(\neg\widehat{Th}(\alpha))$, the Boolean encoding of this tautology.
  - $e\left(\neg\widehat{Th}(\alpha)\right) := \neg e(x{=}y) \vee \neg e(y = z) \vee e(x{=}z)$ --- blocking clause(s)
  - This clause contradicts the current assignment, and hence blocks it from being repeated
  - In general, we denote by t the lemma returned by DP_T.
- →After the blocking clause has been added, the SAT solver is invoked again and suggests another assignment
- →Then invoke DP_T again to check the conjunction of the literals corresponding to the new assignment.
- …

A Basic Lazy Approach



Let B^{i} be the formula B in the i-th iteration of the loop in basic lazy algorithm.
$B^{\mathrm{i}+1}$ is strictly stronger than B^{i} for all $i \geq 1$, because blocking clauses are added but not removed between iterations.

It is not hard to see that this implies that any conflict clause that is learned while solving B^{i} can be reused when solving B^j for $i < j$.

This, in fact, is a special case of **incremental satisfiability**, which is supported by most modern SAT solvers. Hence, invoking an incremental SAT solver can increase the efficiency of the algorithm.

A Basic Lazy Approach: Example

$$\Phi \coloneqq \underbrace{g(a) = c}_{1} \land (\underbrace{f(g(a)) \neq f(c)}_{2} \lor \underbrace{g(a) = d}_{3}) \land \underbrace{c \neq d}_{4}$$

- $\mathrm{PS}_\Phi = y_1 \land (\neg y_2 \lor y_3) \land y_4)$

- $y_1 \colon g(a) = c$
- $y_2 \colon f(g(a)) = f(c)$
- $y_3 \colon g(a) = d$
- $y_4 \colon c = d$

Send$\{1, \bar{2} \lor 3, \bar{4}\}$ to SAT
SAT solver returns model $\{1, \bar{2}, \bar{4}\}$
UF-solver find concretization of $\{1, \bar{2}, \bar{4}\}$ UNSAT
Send $\{1, \bar{2} \lor 3, \bar{4}, \neg(1 \land \bar{2} \land \bar{4})\}$ to SAT
Send $\{1, \bar{2} \lor 3, \bar{4}, \bar{1} \lor 2 \lor 4\}$ to SAT
SAT solver returns model $\{1,3, \bar{4}\}$
UF-solver find concretization of $\{1,3, \bar{4}\}$ UNSAT
Send $\{1, \bar{2} \lor 3, \bar{4}, \bar{1} \lor 2 \lor 4, \bar{1} \lor \bar{3} \lor 4\}$ to SAT
SAT solver returns UNSAT; Original formula is UNSAT in UF

Integration into CDCL

Algorithm 3.3.2: LAZY-CDCL

Input: A formula φ
Output: "Satisfiable" if the formula is satisfiable, and "Unsatisfiable" otherwise

function LAZY-CDCL
    ADDCLAUSES($cnf(e(\varphi))$);
    while (TRUE) **do**
        while (BCP() = "conflict") **do**
            backtrack-level := ANALYZE-CONFLICT();
            if *backtrack-level* < 0 **then return** "Unsatisfiable";
            else BackTrack(*backtrack-level*);
        if $\neg$DECIDE() **then** ▷ Full assignment
            $\langle t, res\rangle$:=DEDUCTION($\hat{Th}(\alpha)$); ▷ α is the assignment
            if *res*="Satisfiable" **then return** "Satisfiable";
            ADDCLAUSES($e(t)$);

Integration into CDCL

Still not clever enough…

- Consider, for example, a formula ϕ that contains literals
  $$x_1 \geq 10,\ \ x_1 < 0,$$
  where x_1 is an integer variable.
- Assume that the Decide procedure assigns $e(x_1 \geq 10) \mapsto \text{true}$ and $e(x_1 < 0) \mapsto \text{true}$. Inevitably, any call to Deduction results in a contradiction between these two facts.

- However, Algorithm Lazy-CDCL does not call Deduction until a full satisfying assignment is found. // waste time to complete the assignment.

Theory Propagation and the DPLL(T) Framework

- Deduction is invoked after BCP stops.
- It finds T-implied literals and communicates them to the CDCL part of the solver in the form of a constraint t.

Such implications are due to the theory T. Accordingly, this technique is known by the name theory propagation. (*When Deduction cannot find an asserting clause t as defined above, t and e(t) are equivalent to true.)

Example.

- Consider the two encoders $e(x_1 \geq 10)$ and $e(_1 < 0)$. After the first of these has been set to true, Deduction detects that $\neg(x_1 < 0)$ is implied.
- In other words, t := $\neg(x_1 \geq 10) \vee \neg(x_1 < 0)$ is T-valid.
- The corresponding encoded (asserting) clause e(t) := $\neg e(x_1 \geq 10) \vee \neg e(x_1 < 0)$
- e(t) is added to B, which leads to an immediate implication of $\neg e(x_1 < 0)$, and possibly further implications.

Algorithm 3.4.1: DPLL(T)

Input: A formula φ
Output: "Satisfiable" if the formula is satisfiable, and "Unsatisfiable" otherwise

```
function DPLL(T)
    AddClauses(cnf(e(φ)));
    while (TRUE) do
        repeat
            while (BCP() = "conflict") do
                backtrack-level := Analyze-Conflict();
                if backtrack-level < 0 then return "Unsatisfiable";
                else BackTrack(backtrack-level);
            ⟨t, res⟩:=Deduction(T̂h(α));
            AddClauses(e(t));
        until t ≡ TRUE;
        if α is a full assignment then return "Satisfiable";
        Decide();
```

Performance, Performance…

- Theory lemmas have to be implied by ϕ and are restricted to a finite set of atoms—typically to ϕ's atoms.
- It is desirable that, when $\neg\widehat{Th}(\alpha)$ is unsatisfiable, e(t) blocks α;
- it is not mandatory, because whether it blocks α or not does not affect correctness—Deduction only needs to be complete when α is a full assignment.
- SMT solvers exploit this fact to perform cheap checks on partial assignments, e.g., bound the time dedicated to them.

Performance, Performance…

- When α is partial, Deduction checks
  - if there is a contradiction on the theory side,
  - and if not, it performs theory propagation.

- For performance, it is frequently better to run an approximation in this step for finding contradictions.
  - This does not change the completeness of the algorithm, since a complete check is performed when α is full.
  - E.g. integer linear arithmetic: Deciding the conjunctive fragment of this theory is NP-complete, and therefore they only consider the real relaxation of the problem, which can be solved in polynomial time.
  - This means that Deduction will sometimes miss a contradiction and hence not return a blocking clause

Performance, Performance…

- Another performance consideration is related to theory propagation, which is required not for correctness, but only for efficiency.
- Exhaustive theory propagation refers to a procedure that finds and propagates all literals that are implied in T by $\widehat{Th}(\alpha)$.

- A simple generic way (called "plunging") to perform theory propagation

  Given an unassigned theory atom at_i , check whether $\widehat{Th}(\alpha)$ implies either at_i or $\neg at_i$.

  The set of unassigned atoms that are checked in this way depends on how exhaustive we want the theory propagation to be.

- In many cases a better strategy is to perform only simple, inexpensive propagations
  - E.g. LIA: to search for simple-to-find implications, such as "if x > c holds, where x is a variable and c a constant, then any literal of the form x > d is implied if d < c"

DPLL(T)

- DPLL(T) algorithm for satisfiability modulo T
  - Extends DPLL (indeed CDCL) algorithm to incorporate reasoning about a theory T
  - Basic Idea:
    - Use CDCL algorithm to find assignments for propositional abstraction of formula

      Use off-the-shelf SAT solver
    - Check the T-satisfiability of assignments found by SAT solver

      Use Theory Solver for T
    - Perform contradiction detection and theory propagation at partial assignments in CDCL

      Use Theory Solver for T

DPLL(T) Example

$(x+1>0 \lor x+y>0) \land (x<0 \lor x+y>4) \land \neg x+y>0$

- DPLL(LIA) algorithm

Invoke DPLL(T) for theory T = LIA (linear integer arithmetic)

DPLL(T) Example

$(x+1>0 \vee x+y>0) \wedge (x<0 \vee x+y>4) \wedge \neg x+y>0$

- DPLL(LIA) algorithm
  - Propagate : $x+y>0 \rightarrow$ false
  - Propagate : $x+1>0 \rightarrow$ true
  - Decide : $x<0 \rightarrow$ true

⇨ Unlike propositional SAT case, we must check T-satisfiability of context

Context

$\neg x+y>0$

$x+1>0$

$x<0^d$

DPLL(T) Example

$(x+1>0 \vee x+y>0) \wedge (x<0 \vee x+y>4) \wedge \neg x+y>0$

- DPLL(LIA) algorithm
  - Propagate : $x+y>0 \rightarrow$ false
  - Propagate : $x+1>0 \rightarrow$ true
  - Decide : $x<0 \rightarrow$ true
  - Invoke theory solver for LIA on: $\{x+1>0, \neg\, x+y>0, x<0\}$

Context is LIA-unsatisfiable! → one of $x+1>0$, $x<0$ must be false

Context

$\neg x+y>0$

$x+1>0$

$x<0^{d}$

DPLL(T) Example

$(x+1>0 \lor x+y>0) \land (x<0 \lor x+y>4) \land \neg x+y>0 \land$
$(\neg x+1>0 \lor \neg x<0)$

Conflicting clause!
...backtrack on a decision

Context

$\neg x+y>0$
$x+1>0$
$x<0^d$

- DPLL(LIA) algorithm
  - Propagate : $x+y>0 \rightarrow$ false
  - Propagate : $x+1>0 \rightarrow$ true
  - Decide : $x<0 \rightarrow$ true
  - Invoke theory solver for LIA on: $\{x+1>0, \neg x+y>0, x<0\}$
    - Add theory lemma $(\neg x+1>0 \lor \neg x<0)$

DPLL(T) Example

(x+1>0 $\vee$ x+y>0) $\wedge$(x<0 $\vee$ x+y>4) $\wedge$ ¬x+y>0 $\wedge$
(¬ x+1>0 $\vee$¬ x<0)

- DPLL(LIA) algorithm
  - Propagate : x+y>0 → false
  - Propagate : x+1>0 → true
  - Propagate : x<0 → false

Context

¬x+y>0
x+1>0
¬ x<0

DPLL(T) Example

(x+1>0 $\vee$ x+y>0) $\wedge$(x<0 $\vee$ x+y>4) $\wedge$ ¬x+y>0 $\wedge$
(¬ x+1>0 $\vee$¬ x<0)

- DPLL(LIA) algorithm
  - Propagate : x+y>0 → false
  - Propagate : x+1>0 → true
  - Propagate : x<0 → false
  - Propagate : x+y>4 → true
  - Invoke theory solver for LIA on: { x+1>0, ¬ x+y>0, ¬x<0， x+y>4 }

Context

¬x+y>0
x+1>0
¬ x<0
x+y>4

DPLL(T) Example

(x+1>0 ∨ x+y>0) ∧(x<0 ∨ x+y>4) ∧ ¬x+y>0 ∧ (¬ x+1>0 ∨¬ x<0)

- DPLL(LIA) algorithm
  - Propagate : x+y>0 → false
  - Propagate : x+1>0 → true
  - Propagate : x<0 → false
  - Propagate : x+y>4 → true
  - Invoke theory solver for LIA on: { x+1>0, ¬ x+y>0, ¬x<0， x+y>4 }

Context is LIA-unsatisfiable! → one of ¬ x+y>0, x+y>4 must be false

Context

¬x+y>0
x+1>0
¬ x<0
x+y>4

DPLL(T) Example

(x+1>0 ∨ x+y>0) ∧(x<0 ∨ x+y>4) ∧ ¬x+y>0 ∧
(¬ x+1>0 ∨¬ x<0) ∧(x+y>0 ∨ ¬ x+y>4)

Conflicting clause!
…no decision to backtrack

- DPLL(LIA) algorithm
  - Propagate : x+y>0 → false
  - Propagate : x+1>0 → true
  - Propagate : x<0 → false
  - Propagate : x+y>4 → true
  - Invoke theory solver for LIA on: { x+1>0, ¬ x+y>0, ¬x<0， x+y>4 }
    - Add theory lemma (x+y>0 ∨ ¬ x+y>4)

Input is LIA-unsat

Context

¬x+y>0
x+1>0
¬ x<0
x+y>4

DPLL(T) Theory Solver

- Input : A set of T-literals M

- Output : either

1. M is T-satisfiable
   - Return model, e.g. { x → 2, y → 3, z → -3, … }

   →Should be *solution-sound*
   - Answers "M is T-satisfiable" only if M is T-satisfiable
2. $\{l_1, \dots, l_n\} \subseteq$ M is T-unsatisfiable // $l_1 \wedge \cdots \wedge l_n$
   - Return conflict clause ($\neg\, l_1 \vee \dots \vee \neg\, l_n$)

   → Should be *refutation-sound*
   - Answers "$\{l_1, \dots, l_n\}$ is T-unsatisfiable" only if $\{l_1, \dots, l_n\}$ is T-unsatisfiable
3. Return lemma

→If solver is solution-sound, refutation-sound, and *terminating*,
- Then it is a *decision procedure* for T

Design of DPLL(T) Theory Solvers

- A DPLL(T) theory solver:
  - Should be solution-sound, refutation-sound, terminating
  - Should produce models when M is T-satisfiable
  - Should produce T-conflicts of minimal size when M is T-unsatisfiable
  - Should be designed to work incrementally
    - M is constantly being appended to/backtracked upon
  - Can be designed to check T-satisfiability either:
    - Eagerly: Check if M is T-satisfiable immediately when any literal is added to M
    - Lazily: Check if M is T-satisfiable only when M is complete
  - Should cooperate with other theory solvers when combining theories
    - (see later)

Outline

- SMT Basis
- Lazy Approach --- DPLL(T)
- Eager Approach --- Bit Blasting

Eager Approach

T-formula → **Encoding** → equisatisfiable propositional formula → **SAT solver** → solution

Perform a full reduction of a T-formula to an equisatisfiable propositional formula in ***one-step***. A ***single run*** of the SAT solver on the propositional formula is then sufficient to decide the original formula.

QFBV State-of-the-art solvers are based on eager approach (a.k.a. *Bit-blasting)*

QFBV

Many compilers have this sort of bug

overflow?

$(x - y > 0) \Leftrightarrow (x > y)$

What is the output? (44)

```
unsigned char number = 200;
number = number + 100;
printf ("Sum: %d\n", number );
```

- Bitwise operator frequently occur in system-level software
  - left-shift, right-shift
  - and, or, xor

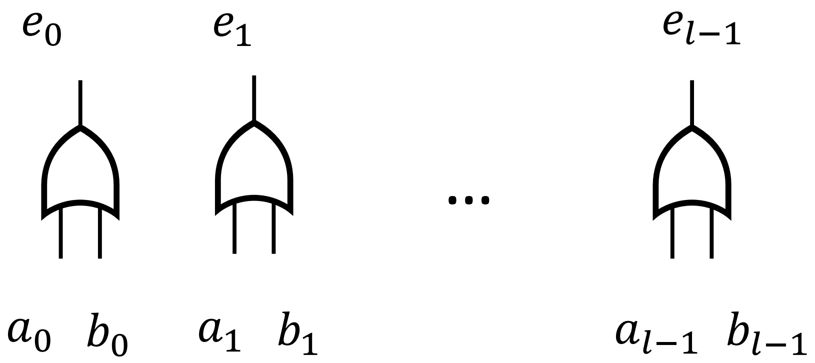
- QFBV Satisfiability is undecidable for an unbounded width, even without arithmetic.
- It is NP-complete otherwise.

Operator to Circuit

Bitwise operators (l-bits): $a|b$

Introduce new bitvector variable e for $a|b$, such that foreach i

$$(a_i \vee b_i) \Leftrightarrow e_i$$

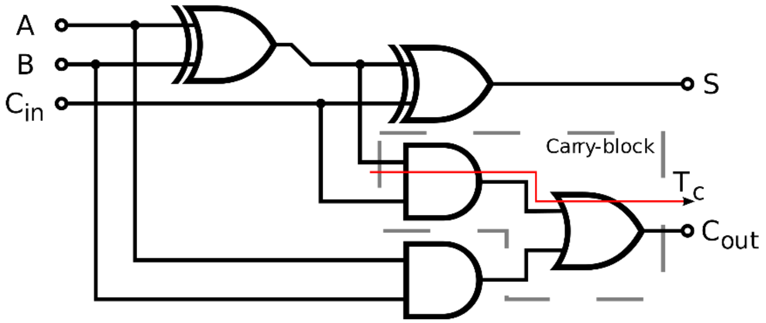


Other bitwise operators is similar

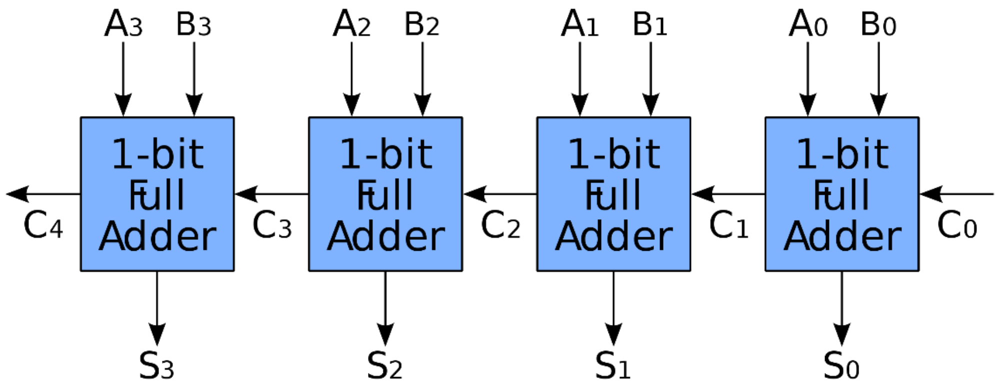
Operator to Circuit

$$a + b$$

one-bit Full adder



four-bits Full adder



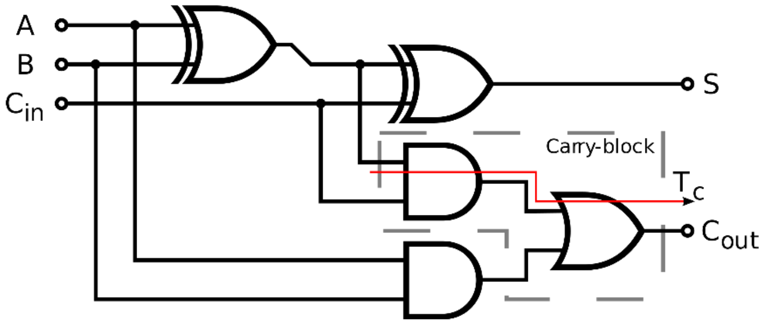
How about 32-bits or 64-bits

Operator to Circuit

$$a - b = (a + \sim b + 1)$$

Complement(补码)
for negative numbers:
$$-b \rightarrow \sim b + 1$$
$\sim b$: invert each bits of b

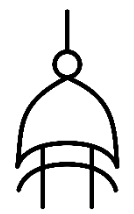
one-bit Full adder




```
6 - 3 ==> 6 + (-3)
  0000 0110 // 6(补码)
+ 1111 1101 // -3(补码)
----------------------
  0000 0011 // 3(补码)
```

Operator to Circuit

$a = b$ $\qquad$ $a_i = b_i \Leftrightarrow e_i$



$\left(a - b = \left(2^l - b\right) + a\right)_{mod\ 2^l}$

If $cout = 1$, then in RHS, the subtract part b is less than the addition part a, i.e. $b < a$

unsigned $a < b$ $\qquad$ $\langle a \rangle_U < \langle b \rangle_U \Leftrightarrow \neg add(a, \sim b, 1).\ cout$

$2 - 3 \Rightarrow 010 - 011 = 010 + 101, cout = 0$

$3 - 2 \Rightarrow 011 - 010 = 011 + 110, cout = 1$

signed $a < b$ $\qquad$ $\langle a \rangle_S < \langle b \rangle_S \Leftrightarrow (a_{l-1} \Leftrightarrow b_{l-1}) \oplus add(a, \sim b, 1).\ cout$

Operator to Circuit

$$a \ll b$$

n-stage (n is the width of b)

stage 1: for each bit i

$$e_i \Leftrightarrow \begin{cases} a_i & : \neg b_0 \\ a_{i-1} & : i \geq 1 \wedge b_0 \\ 0 & : otherwise \end{cases}$$

stage 2: for each bit i

$$e_i' \Leftrightarrow \begin{cases} e_{i-2^1} & : i \geq 2^1 \wedge b_1 \\ e_i & : \neg b_1 \\ 0 & : otherwise \end{cases}$$

...

if ($i < 1$)

$$ite(b_0, (e_i \Leftrightarrow 0), (e_i \Leftrightarrow a_i))$$

if ($i \geq 1$)

$$ite(b_0, (e_i \Leftrightarrow a_{i-1}), (e_i \Leftrightarrow a_i))$$

$$1011011 \ll 101$$

Stage 1:

$$0110110 \Leftarrow 1011011 \ll 001$$

Stage 2:

$$0110110 \Leftarrow 0110110 \ll 000$$

Stage 3:

$$1100000 \Leftarrow 0110110 \ll 100$$

Operator to Circuit

$$a \times b$$

n-stage (shift-and-add):

$$mul(a, b, -1) \doteq 0$$

$(l - 1)$ adder

$$mul(a, b, i) \doteq mul(a, b, i - 1) + (b_i ? (a \ll i) : 0)$$

$$\begin{array}{rl}
1001 & \\
\times\, 0101 & \\
\text{------ –} & \\
1001 & b_0 = 1 \rightarrow a \ll 0 \\
0000\# & b_1 = 0 \rightarrow 0 \\
1001\#\# & b_2 = 1 \rightarrow a \ll 2 \\
0000\#\#\# & b_3 = 0 \rightarrow 0 \\
\text{------- –} &
\end{array}$$

Operator to Circuit

$$a \div b$$

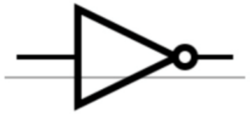
Implemented by adding two constraints:

$$b \neq 0 \Longrightarrow e \times b + r = a,$$
$$b \neq 0 \Longrightarrow r < b$$

If $b = 0$, $a \div b$ is set to a special value.

Circuit to CNF

Tseitin Transformation

Type	Operation	CNF Sub-expression
 AND	$C = A \cdot B$	$(\overline{A} \vee \overline{B} \vee C) \wedge (A \vee \overline{C}) \wedge (B \vee \overline{C})$
 NAND	$C = \overline{A \cdot B}$	$(\overline{A} \vee \overline{B} \vee \overline{C}) \wedge (A \vee C) \wedge (B \vee C)$
 OR	$C = A + B$	$(A \vee B \vee \overline{C}) \wedge (\overline{A} \vee C) \wedge (\overline{B} \vee C)$
 NOR	$C = \overline{A + B}$	$(A \vee B \vee C) \wedge (\overline{A} \vee \overline{C}) \wedge (\overline{B} \vee \overline{C})$
 NOT	$C = \overline{A}$	$(\overline{A} \vee \overline{C}) \wedge (A \vee C)$
 XOR	$C = A \oplus B$	$(\overline{A} \vee \overline{B} \vee \overline{C}) \wedge (A \vee B \vee \overline{C}) \wedge (A \vee \overline{B} \vee C) \wedge (\overline{A} \vee B \vee C)$

Rewrite before Bit-Blasting

n	Number of variables	Number of clauses
8	313	1001
16	1265	4177
24	2857	9529
32	5089	17057
64	20417	68929

Fig. The size of the constraint for an n-bit multiplier expression after Tseitin's transformation

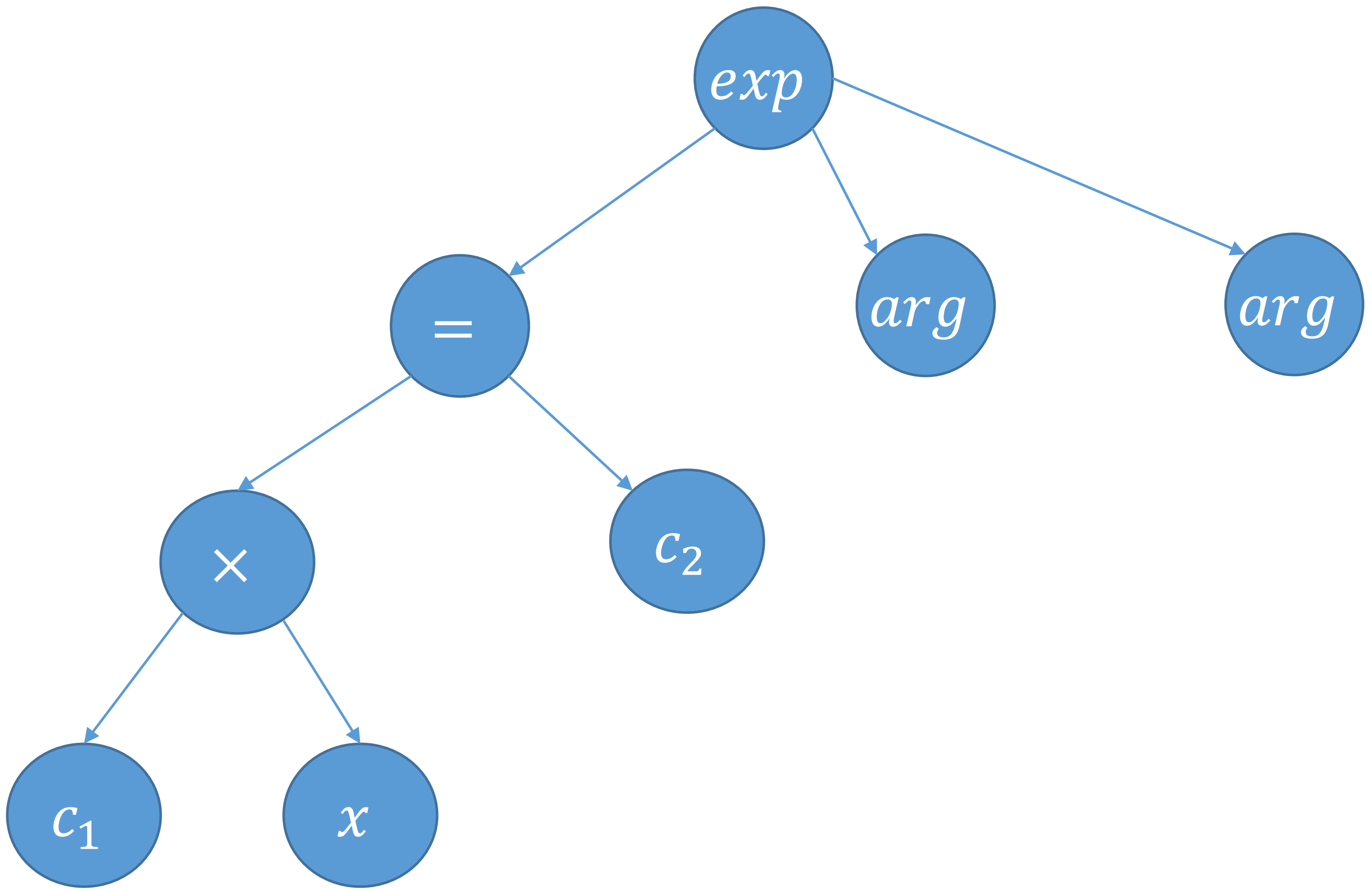
formulas with expensive operators (e.g. multipliers) are often very hard to solve

$$t \times (s \ll (s + t)) \Leftrightarrow s \times (t \ll (s + t))$$

32bits. 10^5 variables.
Can't be solved by CaDiCal within 2 hour

Rewrite before Bit-Blasting

$$c_1 \times x = c_2$$



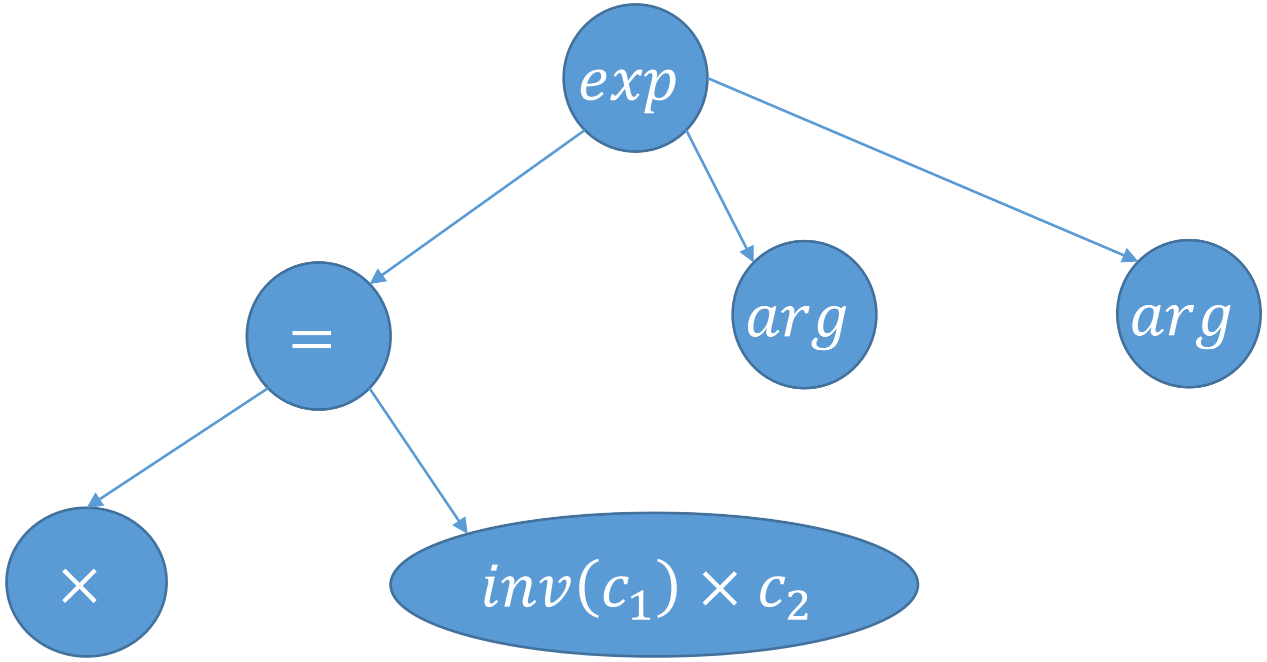
Rewrite before Bit-Blasting

$$c_1 \times x = c_2$$

$$\Downarrow$$

$$x = inv(c_1) \times c_2$$

reduce one multiplier



Deep first order travelling

Theory rewrite rules

- bit2bool (c is 0 or 1)
  - $(ite\ x\ y\ z) = c \ \rightarrow\ (ite\ x\ (y\ = c)\ (z\ = c))$
  - $(not\ x) = c\ \rightarrow x\ =\ (1 - c)$
- mul_eq
  - $cx\ = c' \rightarrow\ x\ =\ c_{inv} \times c'$
  - $cx\ = c'x_2 \rightarrow\ x\ =\ (c_{inv} \times c')\ x_2$
  - …
- mul
  - $cx + c'x \rightarrow (c + c')x$
  - …
- add
  - $(x + (y \ll x)) \rightarrow (x|(y \ll x))$
  - $(x + y \times x) \rightarrow x \times (y + 1)$
- …

Reduce the number of operator

Expensive operator → cheap operator

Propagate const values

- Given an equality $t \ = c$, when c is constant, then replaces t everywhere with c

$t \ = c$

Replace $F(t)$ with $F(c)$

$formula[0]$, $formula[1]$, $formula[2]$, … , $formula[n]$

cyclical scan till fixed

Variable elimination does not always help

$$x = y + z + w$$
$$\ldots (x + z) \ldots$$
$$\ldots (x + 2z) \ldots$$
$$\ldots (x + 3z) \ldots$$
$$\ldots (x + 4z) \ldots$$

$$\Rightarrow$$

$$\ldots (y + 2z + w) \ldots$$
$$\ldots (y + 3z + w) \ldots$$
$$\ldots (y + 4z + w) \ldots$$
$$\ldots (y + 5z + w) \ldots$$

6 adder

8 adder

How to avoid increasing the number of adder and multipliers?

only eliminate variables that occur at most twice

Eliminate unconstrained variables

- a bit-vector function f can be replaced by a fresh bit-vector variable if
  - at least one of its operands is an unconstrained variable v (free variable)
  - f can be "inverted" with respect to v

$$v3 + t = v1 \,\&\, v2$$

If $v1$ and $v2$ are unconstrained variables then no matter what's the value of LHS, it is satisfiable.

$$\Downarrow$$

$$v3 + t = v4$$

If $v3$ is unconstrained variables then no matter what's the value of $v4$ and t, it is satisfiable.

$$\Downarrow$$

$$v5 = v4$$

$$\Downarrow$$

$$v6$$

bv_size_reduction

- Reduce bv size using upper bound and lower bound

$$1 \leq x \leq 4 \ \ (x \text{ has 8 bits})$$

$$\downarrow$$

Replace x with $(concat\ 00000\ x')$

x' is new variable of 3-bits

Local contextual simplification

- bool rewrite

$$\left(or\ args[0] \ldots args\left[num_{args} - 1\right]\right)$$

replace $args[i]$ by $false$ in the other arguments

$$(x\ !=\ 0\ or\ \ y\ =\ x + 1)\ \ ->\ \ (x\ !=\ 0\ or\ y\ =\ 1)$$

Hoist, max sharing

- Reduce the number of adder and multiplier using distribution and association

$$2 \text{ multiplier} + 1 \text{ adder} \rightarrow 1 \text{ multiplier} + 1 \text{ adder}$$

Hoist: $a * b + a * c \rightarrow (b + c) * a$

Max Sharing: $a + (b + c), a + (b + d) \rightarrow (a + b) + c, (a + b) + d$

$$(a + b) \ only \ need \ to \ calculte \ once$$

AIG

AIGs can be used to represent arbitrary boolean formulas and circuits

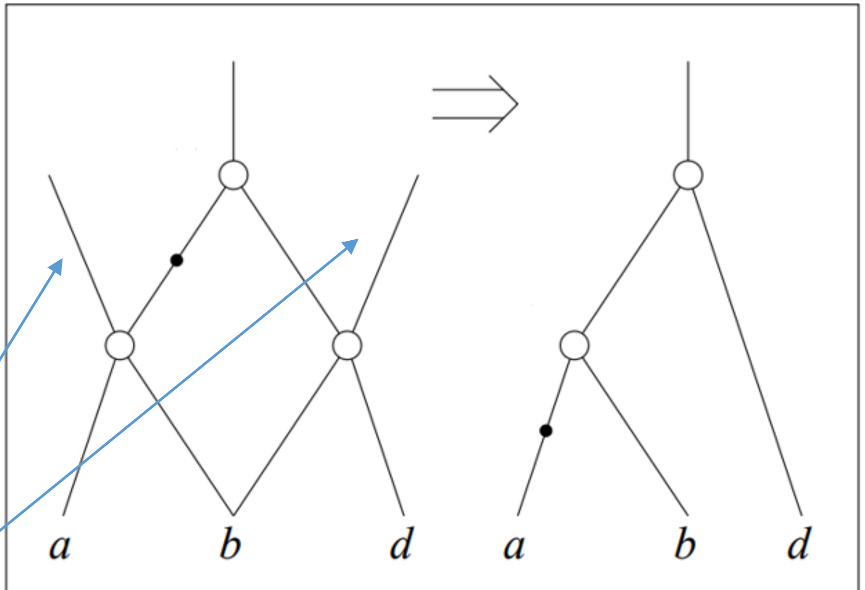
Automatic structure sharing and the simplicity of AIGs make them a compact, simple, easy to use, and scalable representation.

Name	Function	Representation by two-input AND and inversion
Inversion	$\neg x$	$\neg x$
Conjunction	$x \wedge y$	$x \wedge y$
Disjunction	$x \vee y$	$\neg(\neg x \wedge \neg y)$
Implication	$x \rightarrow y$	$\neg(x \wedge \neg y)$
Equivalence	$x \leftrightarrow y$	$\neg(x \wedge \neg y) \wedge \neg(\neg x \wedge y)$
Xor	$x \oplus y$	$\neg(\neg(x \wedge \neg y) \wedge \neg(\neg x \wedge y))$

Table 1. Basic logic operations with two-input AND gates and negation.

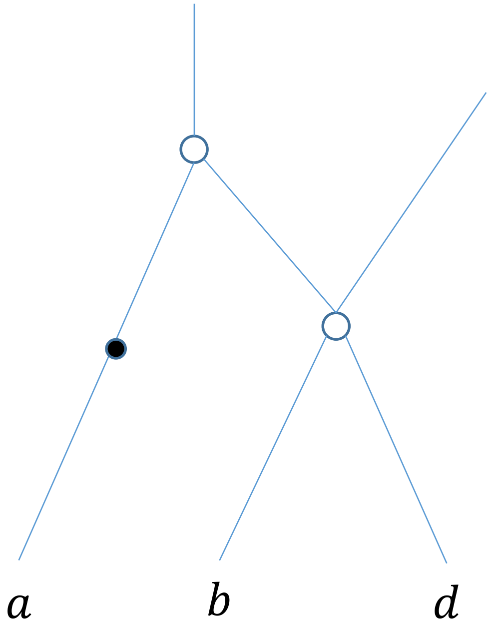
Local 2-level AIG rewrite

2-level:
Consider children and
grand-children



harmful substitution variant

Referenced by other nodes



Locally size decreasing, global non increasing

$$\neg(a \wedge b) \wedge (b \wedge d) \Rightarrow (\neg a \wedge b) \wedge d$$

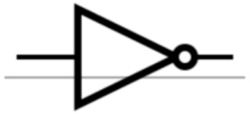
Local 2-level AIG rewrite

Name	LHS	RHS	O	S	Condition
Neutrality	$a \wedge \top$	a	1	S	
Boundedness	$a \wedge \bot$	$\bot$	1	S	
Idempotence	$a \wedge b$	a	1	S	$a = b$
Contradiction	$a \wedge b$	$\bot$	1	S	$a \neq b$
Contradiction	$(a \wedge b) \wedge c$	$\bot$	2	A	$(a \neq c) \vee (b \neq c)$
Contradiction	$(a \wedge b) \wedge (c \wedge d)$	$\bot$	2	S	$(a \neq c) \vee (a \neq d) \vee (b \neq c) \vee (b \neq d)$
Subsumption	$\neg(a \wedge b) \wedge c$	c	2	A	$(a \neq c) \vee (b \neq c)$
Subsumption	$\neg(a \wedge b) \wedge (c \wedge d)$	$c \wedge d$	2	S	$(a \neq c) \vee (a \neq d) \vee (b \neq c) \vee (b \neq d)$
Idempotence	$(a \wedge b) \wedge c$	$a \wedge b$	2	A	$(a = c) \vee (b = c)$
Resolution	$\neg(a \wedge b) \wedge \neg(c \wedge d)$	$\neg a$	2	S	$(a = d) \wedge (b \neq c)$
Substitution	$\neg(a \wedge b) \wedge c$	$\neg a \wedge b$	3	A	$b = c$
Substitution	$\neg(a \wedge b) \wedge (c \wedge d)$	$\neg a \wedge (c \wedge d)$	3	S	$b = c$
Idempotence	$(a \wedge b) \wedge (c \wedge d)$	$(a \wedge b) \wedge d$	4	S	$(a = c) \vee (b = c)$
Idempotence	$(a \wedge b) \wedge (c \wedge d)$	$a \wedge (c \wedge d)$	4	S	$(b = c) \vee (b = d)$
Idempotence	$(a \wedge b) \wedge (c \wedge d)$	$(a \wedge b) \wedge c$	4	S	$(a = d) \vee (b = d)$
Idempotence	$(a \wedge b) \wedge (c \wedge d)$	$b \wedge (c \wedge d)$	4	S	$(a = c) \vee (a = d)$

Table 2. All locally size decreasing, globally non increasing, two-level optimization rules. "O" is the optimization level, "S" the type of symmetry. Subsumption is also known as "Absorption". The condition $a \neq b$ is a short hand for $a = \neg b$ or $b = \neg a$.

Circuit to CNF

Tseitin Transformation

Type	Operation	CNF Sub-expression
 AND	$C = A \cdot B$	$(\overline{A} \vee \overline{B} \vee C) \wedge (A \vee \overline{C}) \wedge (B \vee \overline{C})$
 NAND	$C = \overline{A \cdot B}$	$(\overline{A} \vee \overline{B} \vee \overline{C}) \wedge (A \vee C) \wedge (B \vee C)$
 OR	$C = A + B$	$(A \vee B \vee \overline{C}) \wedge (\overline{A} \vee C) \wedge (\overline{B} \vee C)$
 NOR	$C = \overline{A + B}$	$(A \vee B \vee C) \wedge (\overline{A} \vee \overline{C}) \wedge (\overline{B} \vee \overline{C})$
 NOT	$C = \overline{A}$	$(\overline{A} \vee \overline{C}) \wedge (A \vee C)$
 XOR	$C = A \oplus B$	$(\overline{A} \vee \overline{B} \vee \overline{C}) \wedge (A \vee B \vee \overline{C}) \wedge (A \vee \overline{B} \vee C) \wedge (\overline{A} \vee B \vee C)$

→ SAT solver

Pseudo-Boolean to BV

$$a_1x_1 + a_2x_2 + \cdots + a_nx_n \geq c$$

$$\Downarrow$$

$$lhs \geq rhs$$

$$a_ix_i \Leftrightarrow ite(x_i, bv(a_i), bv(0))$$
$$lhs = bvadd\ (ite_1, ite_2, \dots, ite_n)$$
$$rhs = bv(c)$$

other relation operators (e.g. LT, GT, EQ) can be represent by GE

LIA/NIA to BV

$foreach$ variable x:

1. collect low bound low and upper bound up

2. BV size

If ($low \leq x \leq up$)

$$bits = \log_2(1 + |up - low|)$$

Otherwise

$$bits = num_{bits}$$

num_{bits} = bit_size of abs(Largest constant) + 1

Under approximate
unbound → bound
satisfiability is not preserving

3. BitVector

If (has low)

$$x \Leftrightarrow x_{bv} + low$$

else if (has up)

$$x \Leftrightarrow up - x_{bv}$$

else

$$x \Leftrightarrow x - 2^{bits-1}$$

$(-2^{\text{bits}-1})$ is the *lower bound* of signed int of size *bits*

LIA/NIA to BV

$x \ op \ y$

1. Align BV size of x and y

2. Extend BV size of x and y according to op

$$x_{3bits} + \mathrm{y}_{4\mathrm{bits}}$$

$$\Downarrow$$

$$x_{4bits} + y_{4bits} \qquad\qquad x_{4bits} \times y_{4bits}$$

$$\Downarrow \qquad\qquad \Downarrow$$

$$x_{5bits} + y_{5bits} \qquad\qquad x_{8bits} \times y_{8bits}$$

Feel free to contact me at caisw@ios.ac.cn

Thank you!