hvsat slides

SAT

Boolean Satisfiability

Boolean formulas

- Literals / Variables: $x1$, $x2$, $x3$
- Negation: $\neg x1$
- Conjunction: $x1 \land x2$
- Disjunction: $x1 \lor x2$

Boolean formulas

- CNF: conjunction of disjunctions
  - $(x1 \lor x2 \lor \neg x3) \land (\neg x2 \lor x3)$
- DNF: disjunction of conjunctions
  - $(x2 \land x3) \lor (x1 \land \neg x2) \lor (\neg x2 \land \neg x3)$
- “DNF is easier to think about, but tends to blow up in space”
- “CNF is cheap to construct, but difficult to reason about”

Boolean Satisfiability Problem (SAT)

- Input: boolean formula, usually in CNF
  - Example: (x1 V x2 V ¬x3) Λ (¬x2 V x3)
  - *Literals & Clauses*
- Output: is there a satisfying assignment to all variables that makes the formula hold?
- If there is, return SATISFIABLE and the assignments.
- If there is not, return UNSATISFIABLE.
  - Example: Satisfiable! For instance: x1 = False, x2 = True, x3 = True
  - (x1 V x2 V ¬x3) Λ (¬x2 V x3)

First implementation: Loops

```
for x1 in [True, False]:
    for x2 in [True, False]:
        if formula(x1, x2) return SAT
return UNSAT
```

- Will we return SAT if there is a solution? Yes!
- Will we return UNSAT if there is no solution? Yes!

Implementation: Loops.py

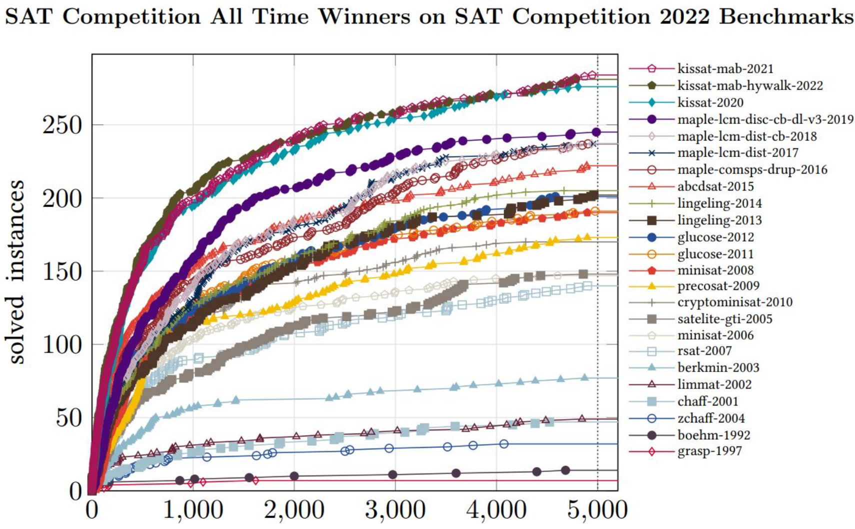
- Avert thine eyes

Why is SAT interesting? (1 / 3)

- It was the first problem proven to be NP-complete.
    - NP-complete: it scales worse than polynomial time
- With n variables, the search space is 2^n
- Other NP-complete problems:
    - Knapsack
    - Hamiltonian Path
    - …
- *SAT is the simplest hard problem*

Why is SAT interesting? (2 / 3)

- SAT is intractable in theory
- But in practice, state-of-the-art SAT solvers can solve huge problem instances in almost no time at all!
- SAT competition: solve as many instances as possible in a given time
- Solvers get faster and faster, due to clever engineering and mathematics!



Why is SAT interesting? (3 / 3)

- The SAT enlightenment: Bounded Model Checking (1999)
- Formal verification technique used mainly in EDA community
- BDDs were used before
  - Grow exponentially in complexity
  - Very dependent on input form
- Previously intractable problems were made possible by SAT
- Important solvers: GRASP and CHAFF

Lingua Franca: DIMACS CNF

- File format for CNF formulas.
- All SAT solvers can parse these.
  - Incredibly easy to parse, actually.

- Comments
- Header (nr variables, nr clauses)
- Clauses (1 means x1, -2 means ¬x2)

```
c simple CNF
c Satisfiable.
p cnf 3 2
1 2 -3 0
-2 3 0
```

Same problem in different formats

$(x1 \lor x2 \lor \neg x3) \land (\neg x2 \lor x3)$

Solution:

Satisfiable, model: [x1 = F, x2 = T, x3 = T]

$(x1 \lor x2 \lor \neg x3) \land (\neg x2 \lor x3)$

```
c simple CNF
c Satisfiable.
p cnf 3 2
1 2 -3 0
-2 3 0
```

```
SATISFIABLE
Final model: [-1, 2, 3]
[ x1,  x2, -x3]
[-x2,  x3]
```

Same problem in different formats

x1 ∧ ¬ x1

Solution:

Unsatisfiable.

```
c simpler CNF
c Unsatisfiable.
p cnf 2 1
1 0
-1 0
```

```
UNSATISFIABLE
Final model: [1]
[ x1]
[-x1]
```

Local search

- AKA “just start flipping bits”
- Assign all variables
- Loop:
    - If SAT return SAT
    - Else negate some assignment

```
for literal in self.literals():
    assignment = self.random_assignment(literal)
    self.assign(assignment)
for _ in range(timeout):
    if self.formula_satisfied():
        return True
    else:
        self.flip_random_assignment()
else:
    logging.critical("naive_local_search timeout")
    return None
```

- Will we return SAT if there is a solution? Yes!
- Will we return UNSAT if there is no solution? No!

Stolen slide 1

1992 - Planning As Satisfiability

$$PAS(S, I, T, G, k) = I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \wedge \bigvee_{i=0}^{k} G(s_i)$$

where:

- S the set of possible states s_i
- I the initial state
- T transitions between states
- G goal state
- k bound

If the formula is satisfiable, then there is a plan of length at most k.

Stolen slide 2

- The decision procedure is very simple to implement and very fast!
- Efficiency depends on which literal to flip, and the values of the parameters.
- Problem with local minima: use of Random Walks!
- Main drawback: incomplete, cannot answer UNSAT!
- Lesson: An agile (fast) SAT solver sometimes better than a clever one!

Implementation: NaiveLocalSearch.py

- Assigns everything completely at random
- No effort to, for example, target falsified clauses

```python
for literal in self.literals():
    assignment = self.random_assignment(literal)
    self.assign(assignment)
for _ in range(timeout):
    if self.formula_satisfied():
        return True
    else:
        self.flip_random_assignment()
else:
    logging.critical("naive_local_search timeout")
    return None
```

Local Search closing words

- Local Search is making a comeback in state-of-the-art solvers
- Algorithms like ProbSAT are almost competitive with… good solvers
- Further optimizations (that I did not implement)
  - Full restarts (reassign all variables)
  - Keep track of how much each variables contribute to conflicts in order to pick variables smarter

DPLL solvers

- Abbreviation of names: Davis-Putnam-Logemann-Loveland
- “Lazy Loop.py”
- Instead of assigning all literals at once, assign one and propagate everything you can.
- If we find a conflict before all variables are assigned, we can avoid assigning more variables!

Unit propagation

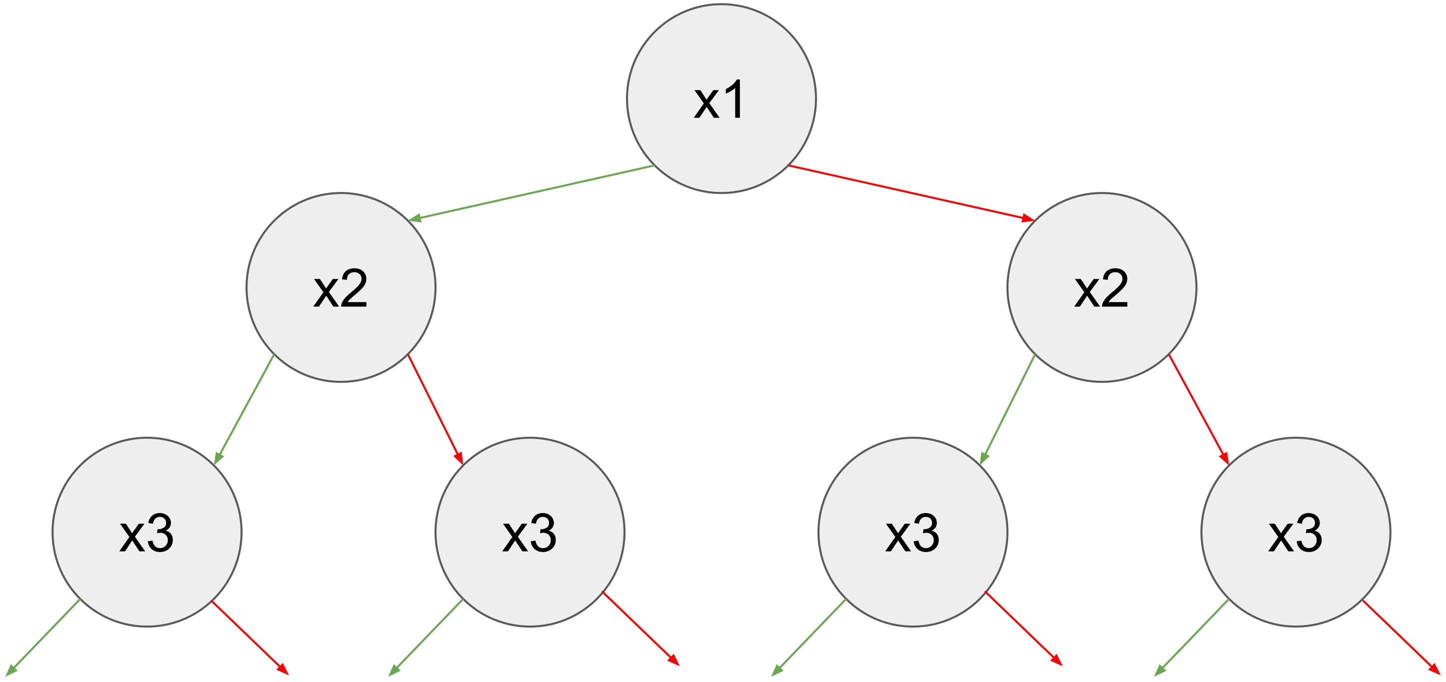
- If a clause is not satisfied and only has one unassigned literal, that literal must be satisfied
- Example:
  - ($x1 \lor \neg x2 \lor x3$)
  - x3 **must** be T to satisfy the clause
- After unit propagation, more clauses may become unit!
- Recursively apply unit propagation until
  - Conflict → UNSAT
  - All clauses satisfied → SAT
  - Nothing can be propagated → Assign some new literal

Implementation: DPLL.py

```python
def DPLL(self, model0):
    sat, model1 = self.unit_propagation(model0)
    if sat is not None:
        return sat, model1
    # Pick an unassigned literal
    assigned_literals = list(map(abs, model1))
    unassigned_literals = [lit for lit in self.literals()
                           if lit not in assigned_literals]
    picked_literal = unassigned_literals[0]

    # Set it to True and recurse
    modelTrue0 = self.assign(model1, picked_literal)
    sat, modelTrue1 = self.DPLL(modelTrue0)
    if sat:
        return sat, modelTrue1

    # Set it to False and recurse
    modelFalse0 = self.assign(model1, -picked_literal)
    return self.DPLL(modelFalse0)
```



- Loops: search models at the leaves
- DPLL: descend slowly through the tree and check if the current model is already SAT/UNSAT

CDCL solvers

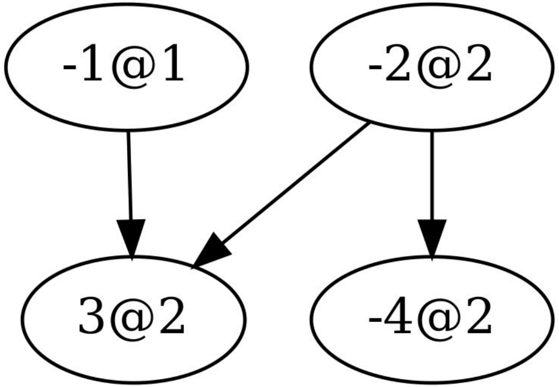
- Conflict Driven Clause Learning
- State of the art
- Like DPLL, but when you find a conflict you figure out what caused it and add a new clause to avoid it
    - Clause learning!
- Introducing shortcuts in the search tree
- Minisat is a CDCL solver

Decision level

- Decision level is the number of literals that have been arbitrarily assigned
- Notation:
  - 1 @ 3: variable 1 (x1) is assigned True at decision level 3
  - -5 @ 2: variable 5 (x5) is assigned False at decision level 2
- If we have a conflict at decision level 0:
  - We have assigned nothing, yet we have a conflict ⇒ return UNSAT
- Our CDCL implementation is not recursive (which the DPLL implementation is), so we need to track this.

Implication Graph

- We need to track **why** a particular literal was assigned and **when**
  - When: Decision level
  - Why: Implications
- Literal assignment can be:
  - **arbitrary** (once per decision level, “pick an unassigned variable”) or
  - **forced** (unit propagation)
- The implication graph tracks both **why** and **when**



Implication graph / CDCL example

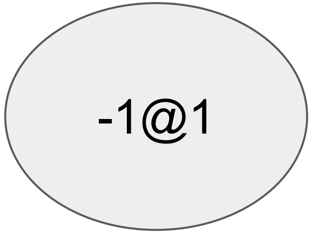
- $(x1 \vee x2 \vee x3) \wedge (x2 \vee \neg x4) \wedge (x2 \vee \neg x3 \vee x4)$

Implication graph / CDCL example

- $(x1 \lor x2 \lor x3)$
  $(x2 \lor \neg x4)$
  $(x2 \lor \neg x3 \lor x4)$

Implication graph / CDCL example

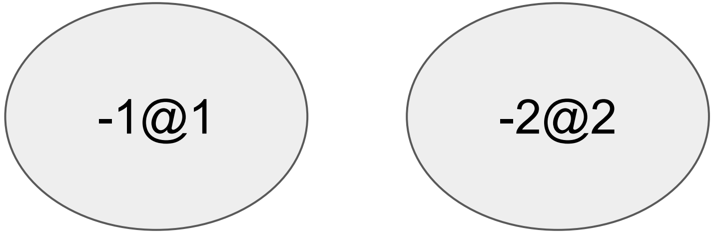
- (x1 V x2 V x3)
  (x2 V ¬x4)
  (x2 V ¬x3 V x4)



arbitrary assignment

Implication graph / CDCL example

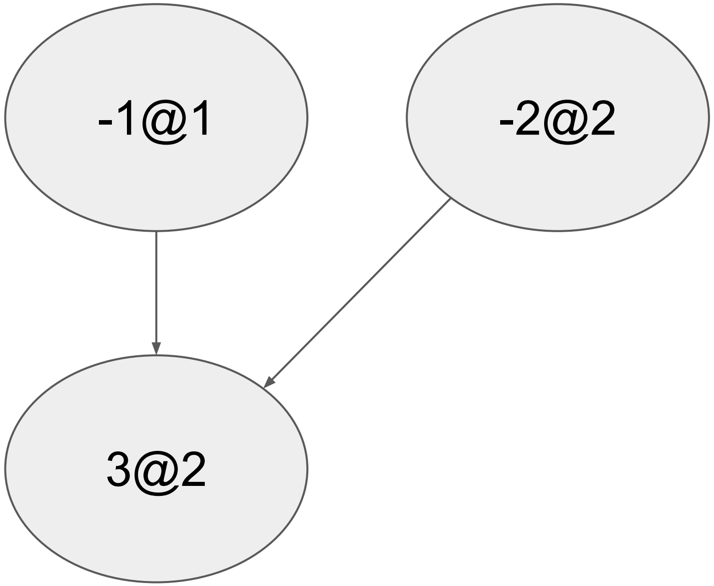
- $(x_1 \lor x_2 \lor x_3)$
  $(x_2 \lor \neg x_4)$
  $(x_2 \lor \neg x_3 \lor x_4)$



arbitrary assignment

Implication graph / CDCL example

- (x1 V x2 V x3)
  (x2 V ¬x4)
  (x2 V ¬x3 V x4)

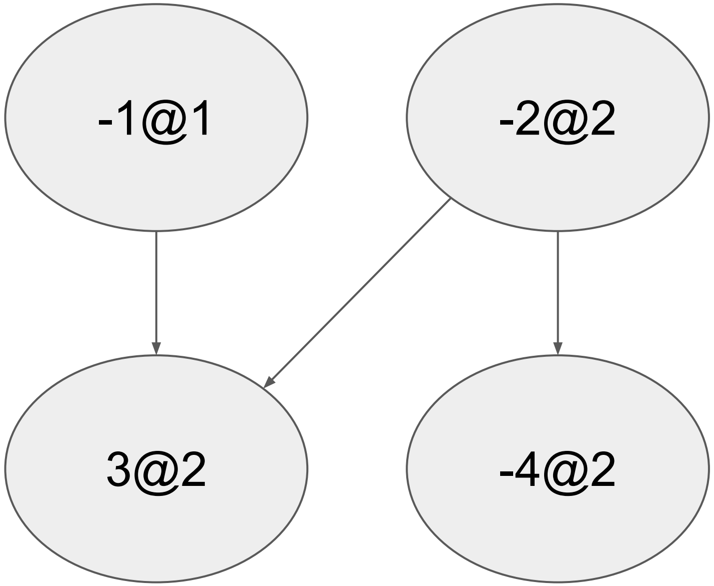


unit propagation

Implication graph / CDCL example

- (x1 V x2 V x3)
  (x2 V ¬ x4)
  (x2 V ¬ x3 V x4)

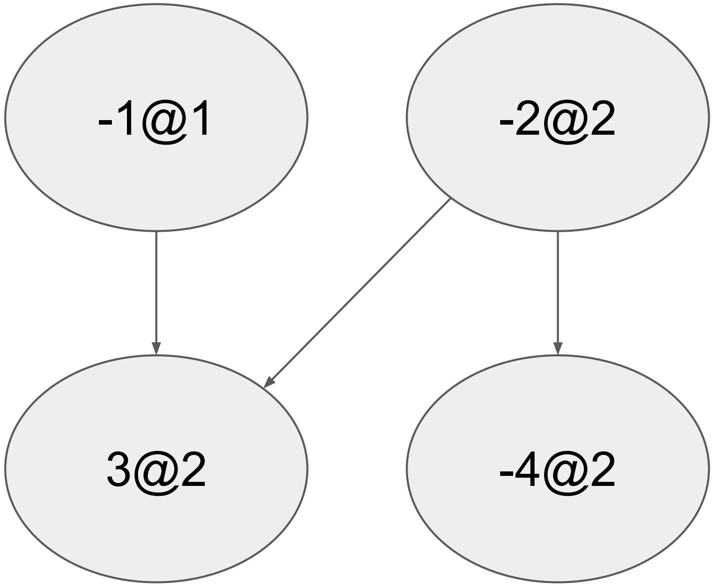
unit propagation



Implication graph / CDCL example

- $(x1 \lor x2 \lor x3)$
  $(x2 \lor \neg x4)$
  $(x2 \lor \neg x3 \lor x4)$

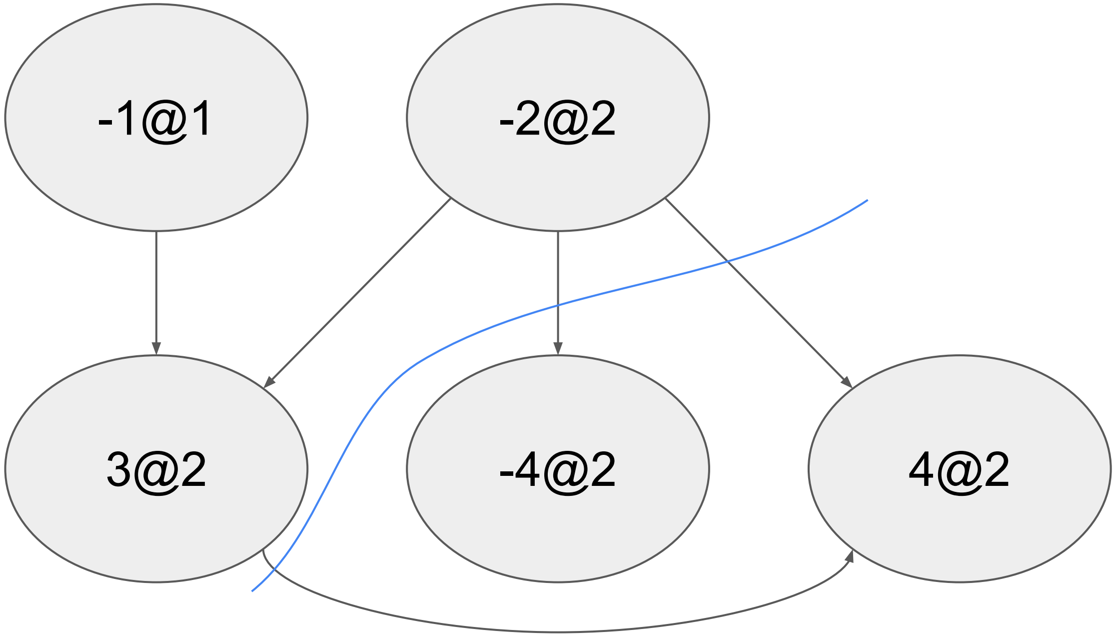
Conflict! Why?



Implication graph / CDCL example

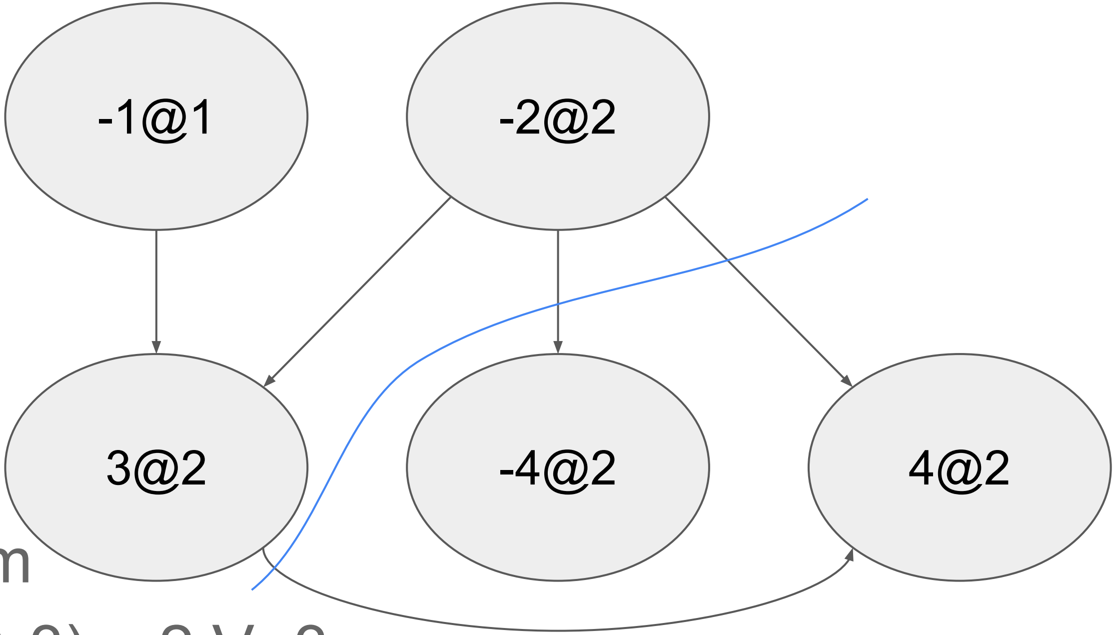
- (x1 V x2 V x3)
  (x2 V ¬x4)
  (x2 V ¬x3 V x4)

Conflict! Why?



Implication graph / CDCL example

- (x1 V x2 V x3)
  (x2 V ¬x4)
  (x2 V ¬x3 V x4)

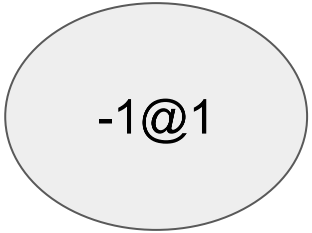


-2 Λ 3 is the problem

New clause: ¬(-2 Λ 3) = 2 V -3

Implication graph / CDCL example

- (x2 V ¬x3)
  (x1 V x2 V x3)
  (x2 V ¬x4)
  (x2 V ¬x3 V x4)

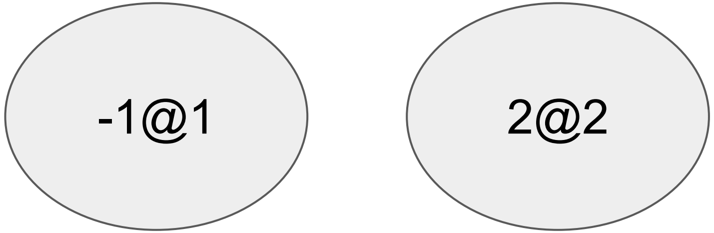


new clause in place!
roll back and reassign x2

Implication graph / CDCL example

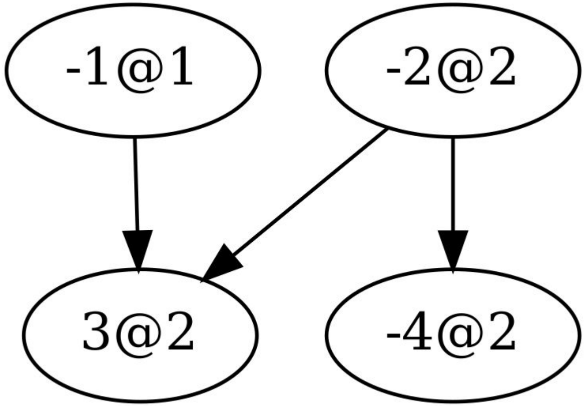
- (x2 V ¬x3)
  (x1 V x2 V x3)
  (x2 V ¬x4)
  (x2 V ¬x3 V x4)

  SAT!
  We did not need the learned clause this time.
  The implication graph is needed for conflict analysis.



Implementation: CDCL.py

- The model is tracked by a custom ImplicationGraph data structure, which can be visualized because fun
- Essentially a list of [dstnode, [srcnode]]
- Unit propagation is essentially unchanged
- Conflict analysis was tricky, but straightforward once you know what to do
- Simple restart algorithm



Out of scope SAT things

- Watched Literals
  - Lazy pointer data structure, for avoiding searching the entire clause
  - Lazy: do nothing upon backjumping
- Variable State Independent Decaying Sum
  - Lazy data structure for choosing the next literal to assign
- SAT/UNSAT mode
  - SAT and UNSAT instances benefit from different techniques
- Any intelligent choice of next variable
- Real-world examples

SAT take-aways

- It’s the simplest hard problem
    - Other hard problems can be formalized as SAT, and then solved efficiently
    - Or rather, in theory just as hard, but efficiently in practice most of the time
- State of the art solvers are crazy good
    - And just get faster and faster due to clever algorithms and heuristics
- Modern solvers are CDCL-based
- BMC was the “killer app” for SAT

SMT

Satisfiability Modulo Theories

SMT: Satisfiability Modulo Theories

- SAT with richer logics (Theories) on top, such as
- EUF: Equality and Uninterpreted Functions
  - $x1 = x2 \Rightarrow f(x1) = f(x2)$
  - Workhorse: Congruence Closure
- LIA: Linear Integer Arithmetic
  - $(x1 \geq 0) \land (x1 \leq x2)$
  - Workhorse: Simplex
- Arrays
- Combining theories
- Quantifiers

Example with EUF and LIA

- $(x1 \geq 0) \land (x1 < 1) \land ((x2 = x1) \lor (x2 = 0)) \land (f(x1)=f(x2) \Rightarrow \neg(g(x1)=g(x2)))$
- $x1 = 0$
- $x2 = 0$
- $x1 = x2$
- $f(x1) = f(x2)$
- $g(x1) = g(x2)$
- $\neg(g(x1)=g(x2))$
- → UNSAT

Lingua Franca: smtlib

```
(set-logic QF_LRA) ; Reals
(declare-const a Real)
(declare-const b Real)
(declare-const c Real)
(declare-const d Real)
(assert (= 1 (+ (* 2 a) c))) ; 2a + c = 1
(assert (= 0 (+ (* 2 b) d))) ; 2b + d = 0
(assert (= 0 (+ (* 2 c) a))) ; a + 2c = 0
(assert (= 1 (+ (* 2 d) b))) ; b + 2d = 1
(check-sat)
(get-model)
```

Python interface

```
%reset -f
from z3 import *
M = [[2, 1],
     [1, 2]]
# Declare four variables over the reals to represent the inverse matrix M^-1
a, b, c, d = Reals("a b c d")
Minv = [[a, b],
        [c, d]]
# Formulate four equations characterising the inverse, use the function solve
eq1 = M[0][0]*Minv[0][0] + M[0][1]*Minv[1][0] == 1
eq2 = M[0][0]*Minv[0][1] + M[0][1]*Minv[1][1] == 0
eq3 = M[1][0]*Minv[0][0] + M[1][1]*Minv[1][0] == 0
eq4 = M[1][0]*Minv[0][1] + M[1][1]*Minv[1][1] == 1
solve(And(eq1, eq2, eq3, eq4))
```

```
[c = -1/3, a = 2/3, d = 2/3, b = -1/3]
```

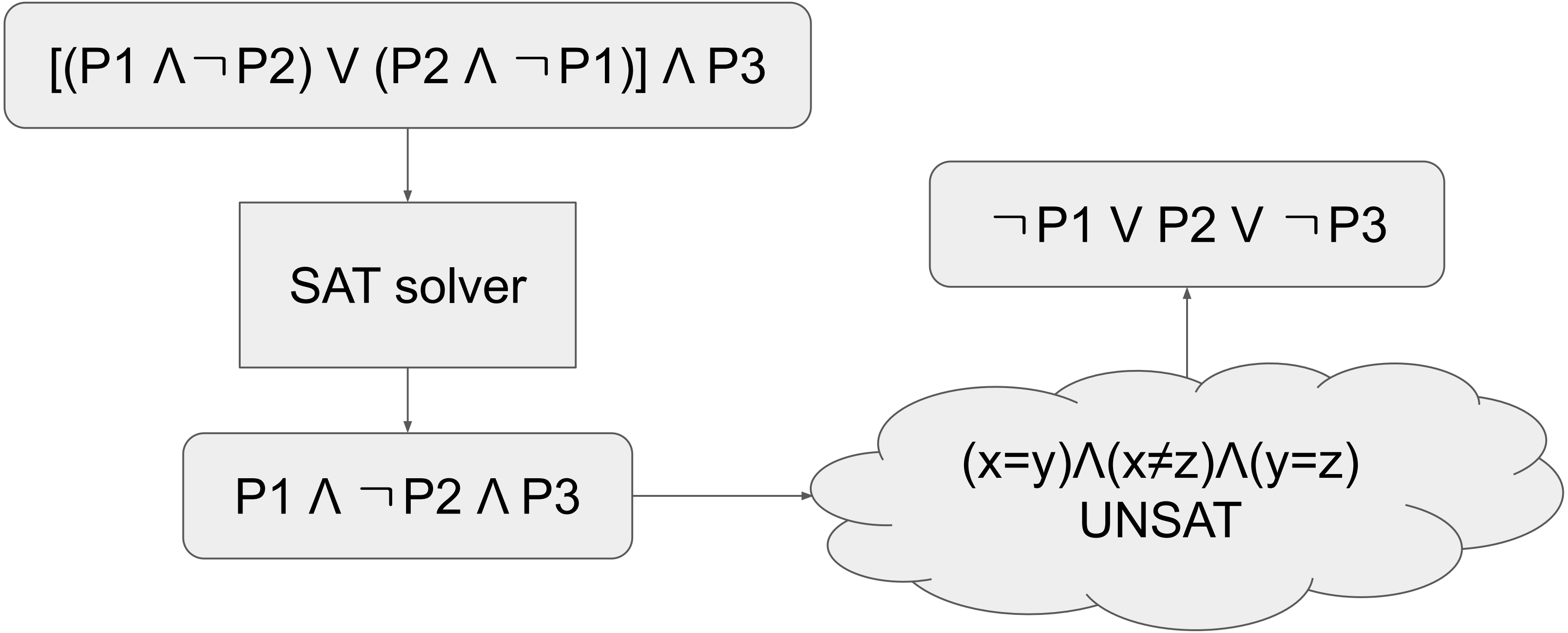
Lazy SMT

- Principle: *deciding satisfiability of a formula can be reduced to deciding the theory satisfiability of conjunctions of constraints.*
- Example: $[(x=y)\wedge(x\neq z) \vee (x=z)\wedge(x\neq y)] \wedge (y=z)$
- Assign $P1 := x=y$ $P2 := x=z$ $P3 := y=z$
- Formula: $[(P1 \wedge \neg P2) \vee (P2 \wedge \neg P1)] \wedge P3$
- DNF: $(P1 \wedge \neg P2 \wedge P3) \vee (\neg P1 \wedge P2 \wedge P3)$
  - Check each conjunction separately (theory satisfiability)
  - $(P1 \wedge \neg P2 \wedge P3)$: $(x=y)\wedge(x\neq z)\wedge(y=z)$
  - $(\neg P1 \wedge P2 \wedge P3)$: $(x\neq y)\wedge(x=z)\wedge(y=z)$
- All conjuncts are UNSAT, so the expression is UNSAT.

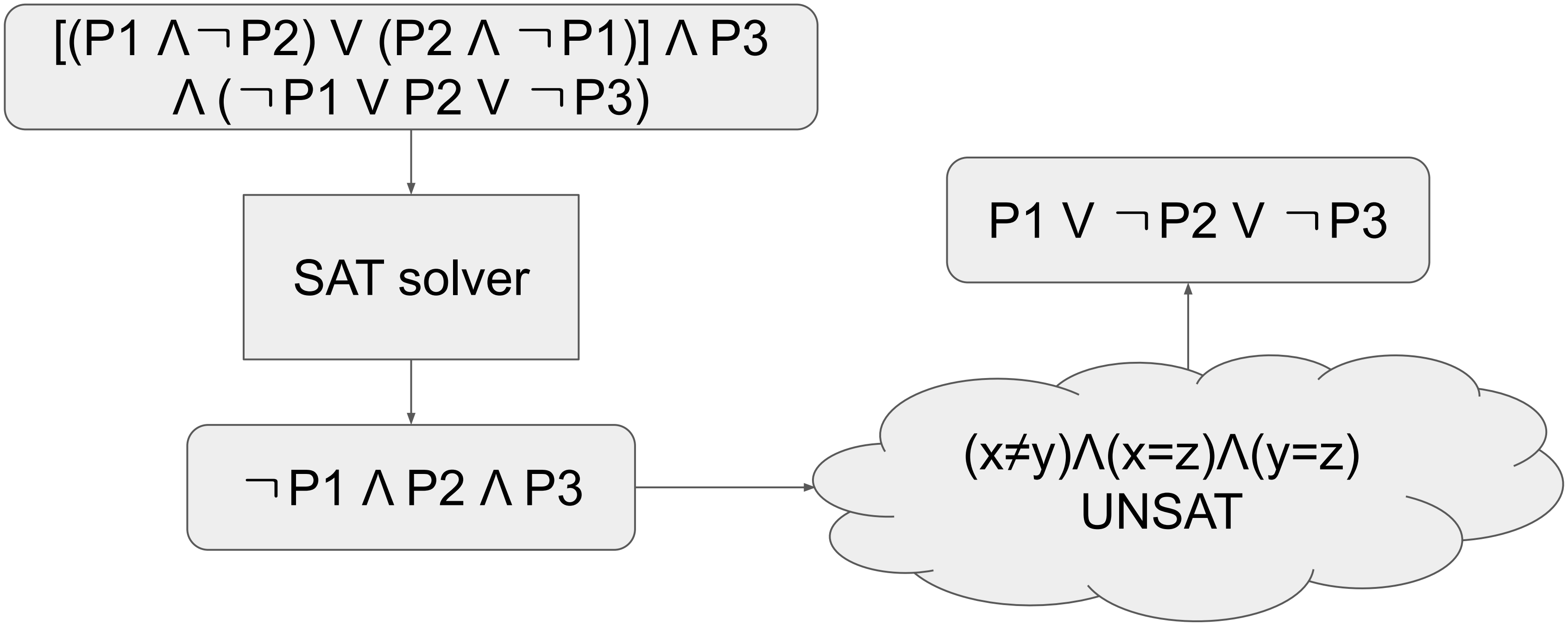
Lazy SMT

- Principle: *deciding satisfiability of a formula can be reduced to deciding the theory satisfiability of conjunctions of constraints.*
- **Converting to DNF is too expensive.** Can we avoid it?
- Lazy SMT: use a SAT solver to enumerate conjuncts!

Lazy SMT


[(P1 Λ ¬ P2) V (P2 Λ ¬ P1)] Λ P3
SAT solver
P1 Λ ¬ P2 Λ P3
(x=y)Λ(x≠z)Λ(y=z)
UNSAT
¬ P1 V P2 V ¬ P3

Lazy SMT



Lazy SMT

$$[(P1 \land \neg P2) \lor (P2 \land \neg P1)] \land P3$$
$$\land (\neg P1 \lor P2 \lor \neg P3)$$
$$\land (P1 \lor \neg P2 \lor \neg P3)$$

↓

SAT solver

↓

UNSAT

CDCL(T)

- Lazy SMT: modular SAT and T solvers, “offline”
- CDCL(T): “online” approach, tighter integration between solvers
  - What should you give T solvers?
  - Can T solvers prune the search space?
  - Propagation?
  - …
- Out of scope for today

The eager approach, “bit blasting”

- Convert as much as possible into pure SAT
- Usually the best and fastest way, especially for bit vector arithmetic

Implementation project idea

- Choose the simplest logic
  - EUF
- Implement a solver for that logic
  - Congruence Closure
- Use our SAT solver to find conjunctions
- ⇒ Lazy SMT

EUF: Equality and Uninterpreted Functions

EUF syntax

(Formula) $\varphi ::= Atom \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \neg\varphi$
(Atom) $Atom ::= Term = Term$
(Term) $Term ::= Var \mid Const \mid F(Term)$

Reflexivity: $\dfrac{}{E = E}$ Transitive: $\dfrac{E_1 = E_2 \qquad E_2 = E_3}{E_1 = E_3}$

Symmetry: $\dfrac{E_2 = E_1}{E_1 = E_2}$ Congruence: $\dfrac{E_1 = E_2}{f(E_1) = f(E_2)}$

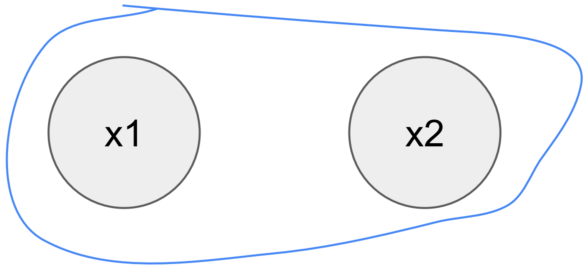
EUF examples

- $x1 = x2$
  - SAT
- $(x1 = x2) \land (F(x1) \neq F(x2))$
  - UNSAT
- $(x1 = x2) \land (x2 = x3) \land (G(x1) \neq G(x3))$
  - UNSAT
- $(x1=x2 \lor x3=x4) \land \neg(F(x1)=F(x2) \lor G(x3)=G(x4))$
  - UNSAT

Congruence Closure

Congruence Closure

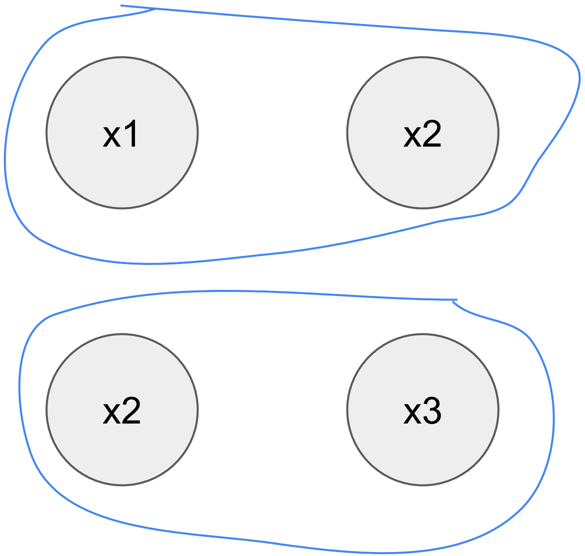
- x1 = x2



New equivalence class

Congruence Closure

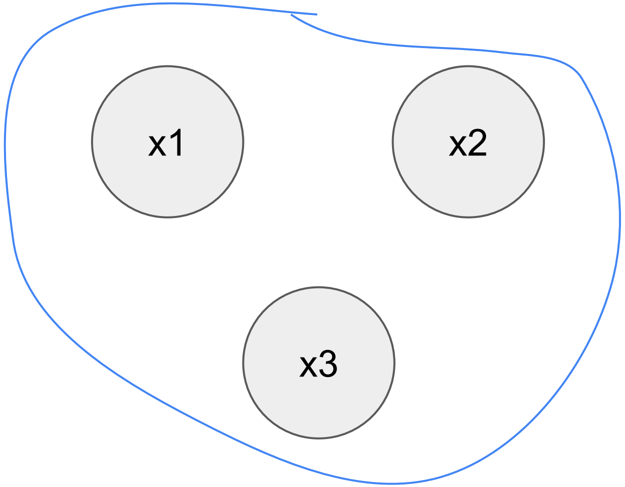
- x1 = x2
- x2 = x3



New equivalence class

Congruence Closure

- x1 = x2
- x2 = x3

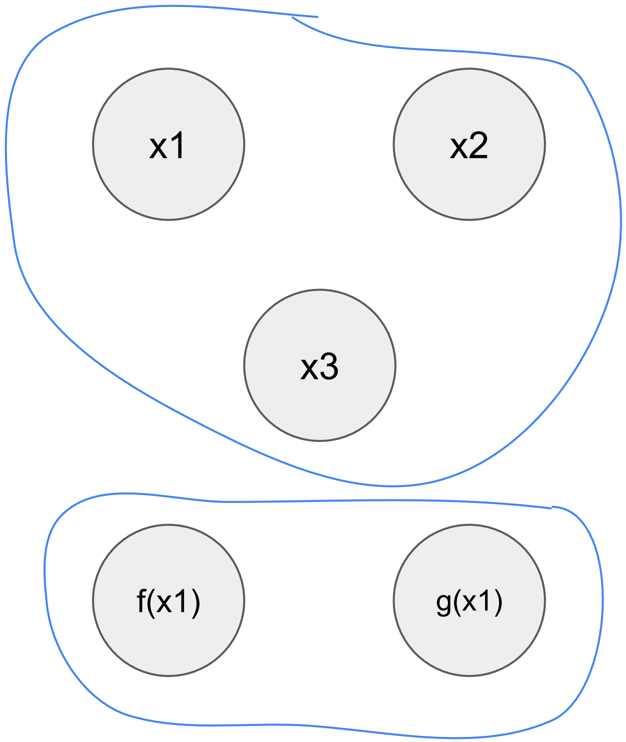


If any classes share a term, merge them

Congruence Closure

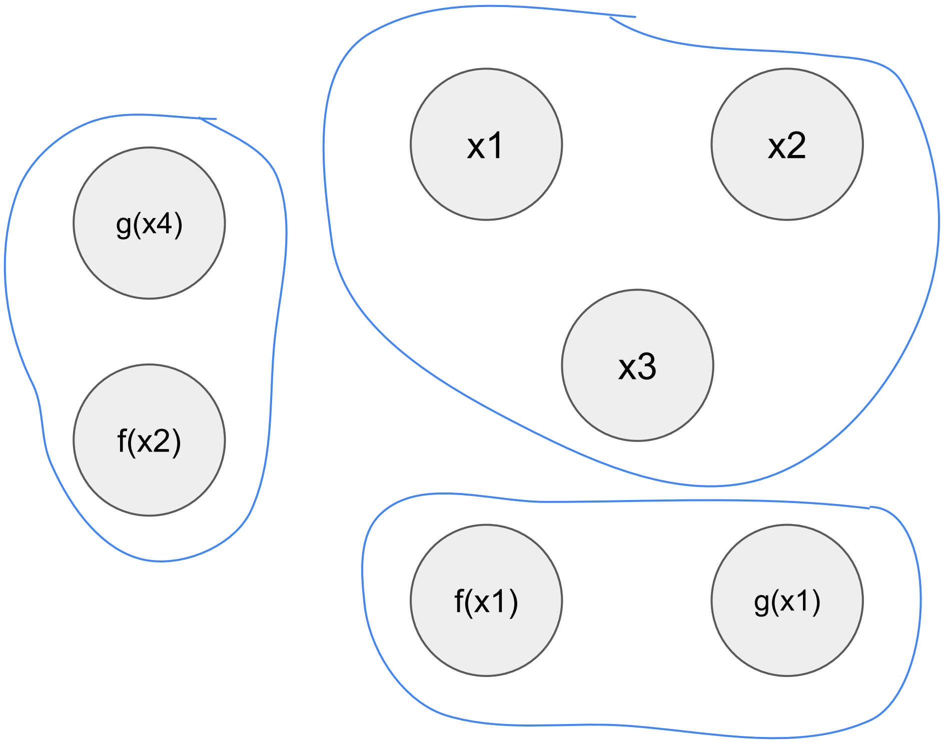
- x1 = x2
- x2 = x3
- f(x1) = g(x1)

New equivalence class



Congruence Closure

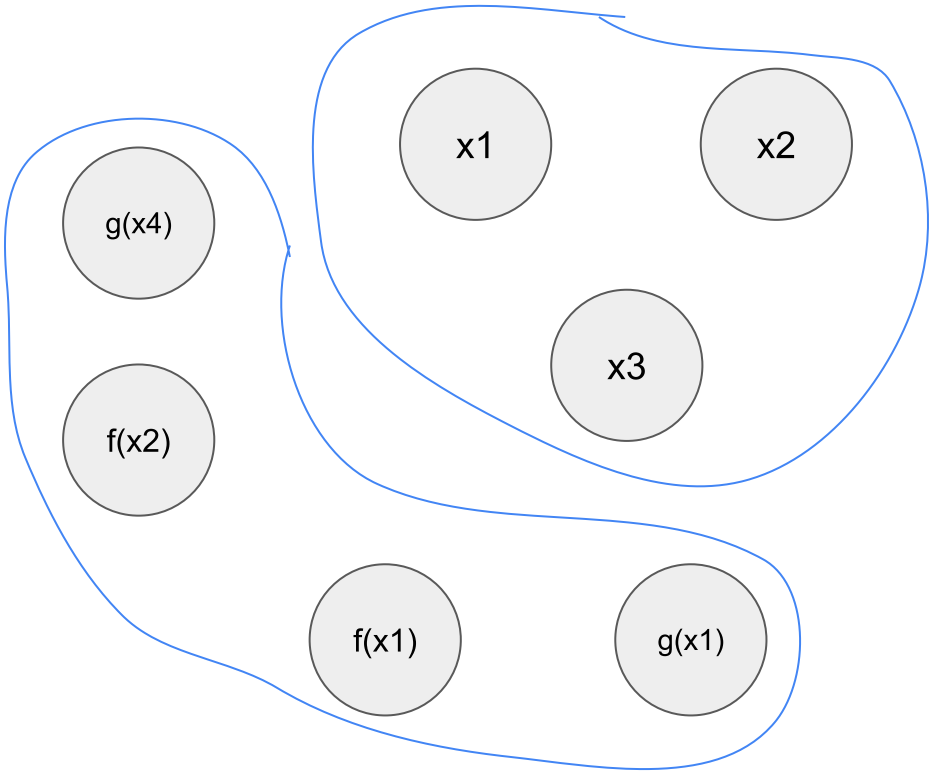
- x1 = x2
- x2 = x3
- f(x1) = g(x1)
- f(x2) = g(x4)



New equivalence class

Congruence Closure

- $x1 = x2$
- $x2 = x3$
- $f(x1) = g(x1)$
- $f(x2) = g(x4)$



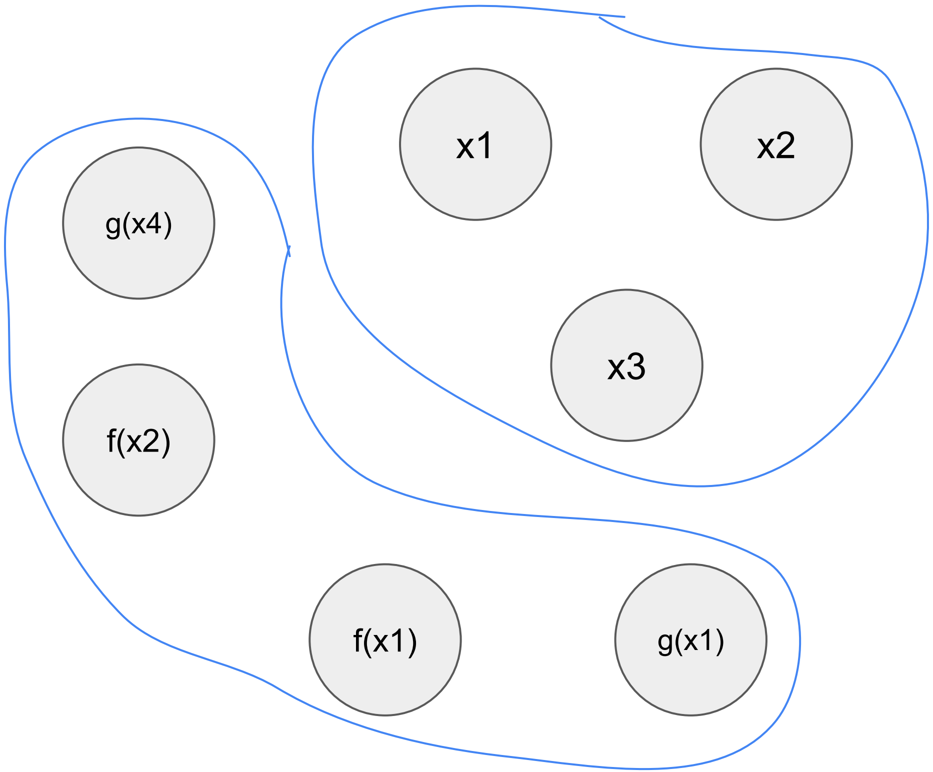
Congruence:

$x1=x2$ so $f(x1)=f(x2)$

Congruence Closure

- x1 = x2
- x2 = x3
- f(x1) = g(x1)
- f(x2) = g(x4)
- f(x1) ≠ x1

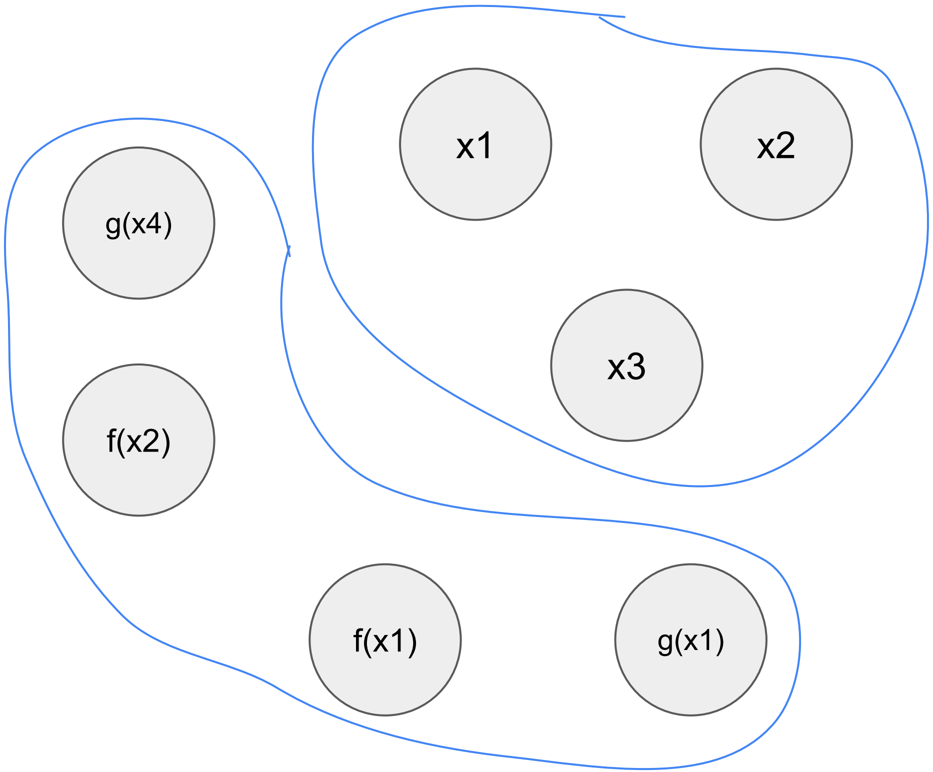
SAT, not in same class



Congruence Closure

- x1 = x2
- x2 = x3
- f(x1) = g(x1)
- f(x2) = g(x4)
- f(x1) ≠ x1
- f(x1) ≠ g(x4)

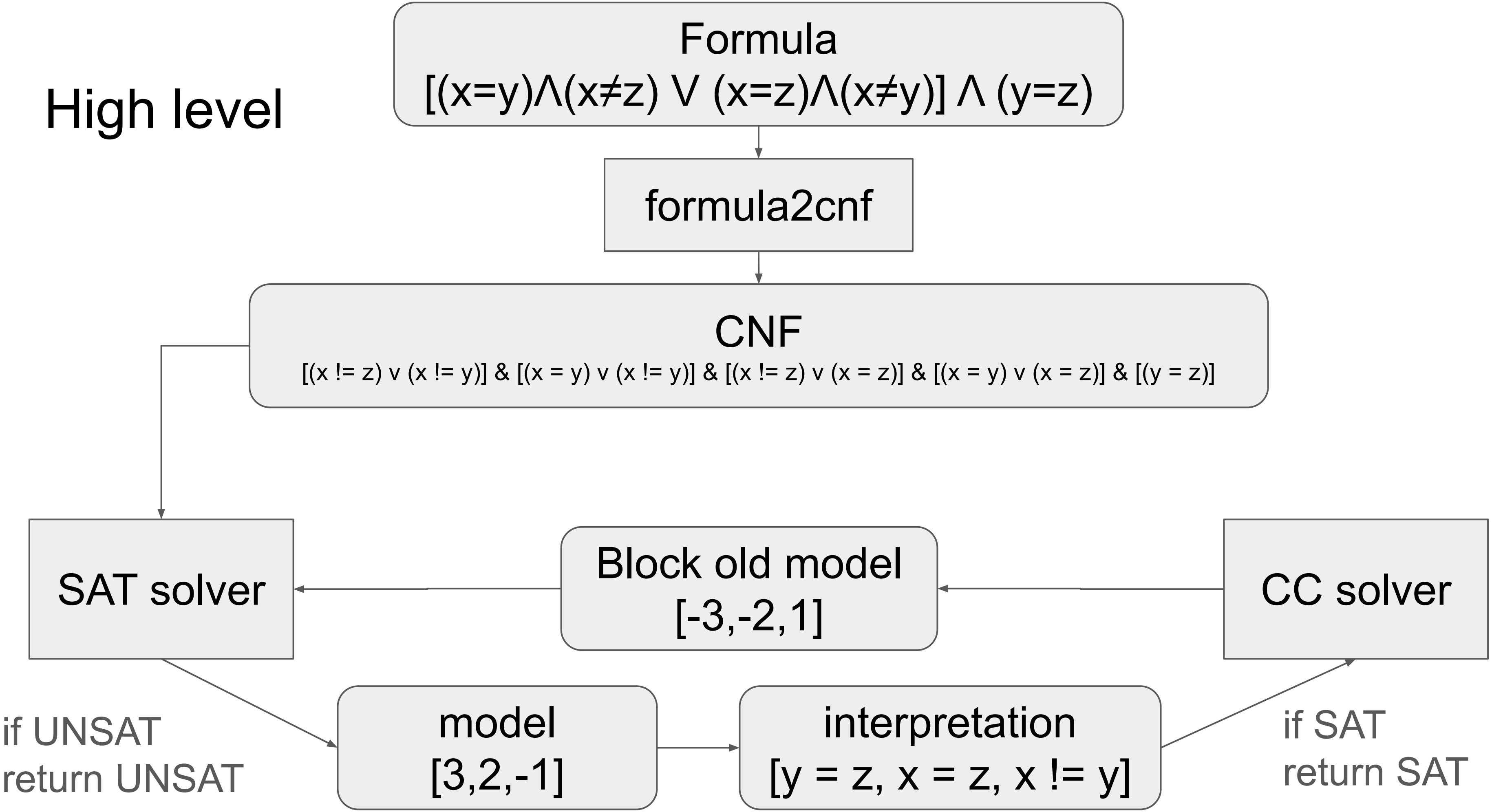
UNSAT, f(x1) = g(x4)



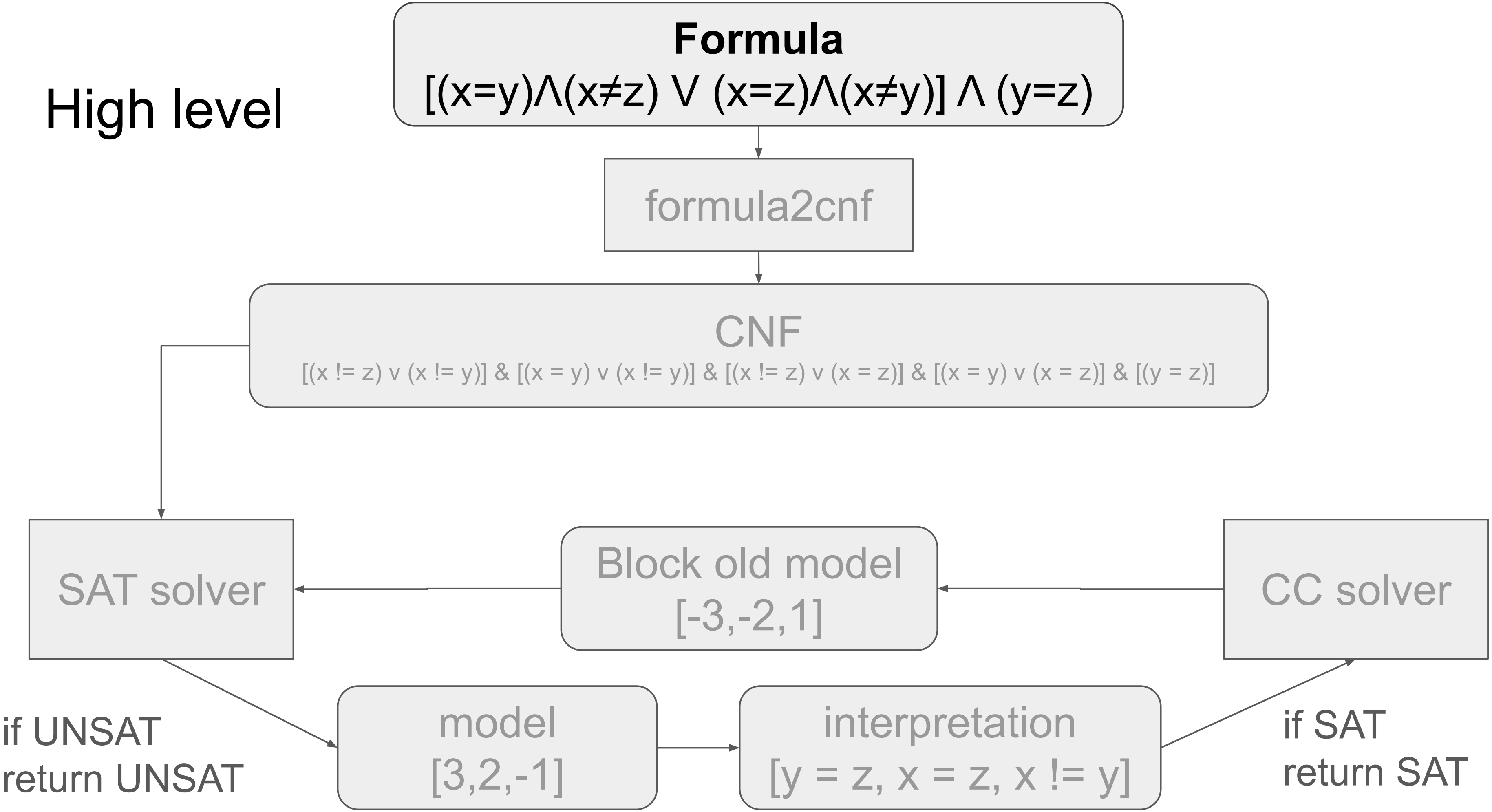
Implementation: CongruenceClosure.py

- This was intended to be a small thing
- 1000+ lines of python later, that turned out not to be the case

High level


Formula
[(x=y)Λ(x≠z) V (x=z)Λ(x≠y)] Λ (y=z)
formula2cnf
CNF
[(x != z) v (x != y)] & [(x = y) v (x != y)] & [(x != z) v (x = z)] & [(x = y) v (x = z)] & [(y = z)]
SAT solver
Block old model
[-3,-2,1]
CC solver
if UNSAT
return UNSAT
model
[3,2,-1]
interpretation
[y = z, x = z, x != y]
if SAT
return SAT

High level


Formula
[(x=y)Λ(x≠z) V (x=z)Λ(x≠y)] Λ (y=z)
formula2cnf
CNF
[(x != z) v (x != y)] & [(x = y) v (x != y)] & [(x != z) v (x = z)] & [(x = y) v (x = z)] & [(y = z)]
SAT solver
Block old model
[-3,-2,1]
CC solver
if UNSAT
return UNSAT
model
[3,2,-1]
interpretation
[y = z, x = z, x != y]
if SAT
return SAT

Formula

EUF syntax

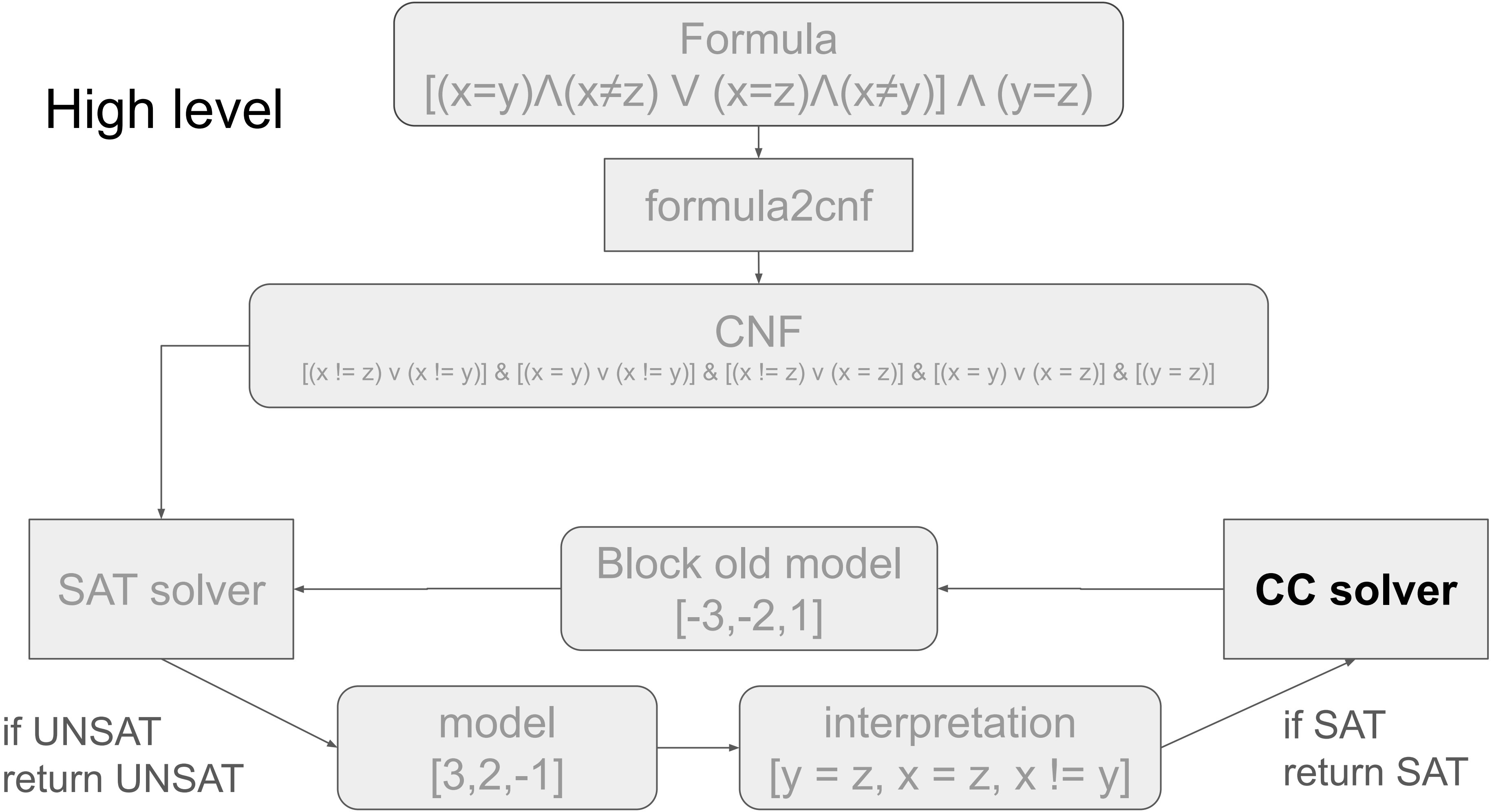
(Formula) $\varphi ::= Atom \mid \varphi \land \varphi \mid \varphi \lor \varphi \mid \neg\varphi$

(Atom) $Atom ::= Term = Term$

(Term) $Term ::= Var \mid Const \mid F(Term)$

- General formulas for EUF

High level



Formula
[(x=y)Λ(x≠z) V (x=z)Λ(x≠y)] Λ (y=z)
formula2cnf
CNF
[(x != z) v (x != y)] & [(x = y) v (x != y)] & [(x != z) v (x = z)] & [(x = y) v (x = z)] & [(y = z)]
SAT solver
Block old model
[-3,-2,1]
CC solver
if UNSAT
return UNSAT
model
[3,2,-1]
interpretation
[y = z, x = z, x != y]
if SAT
return SAT

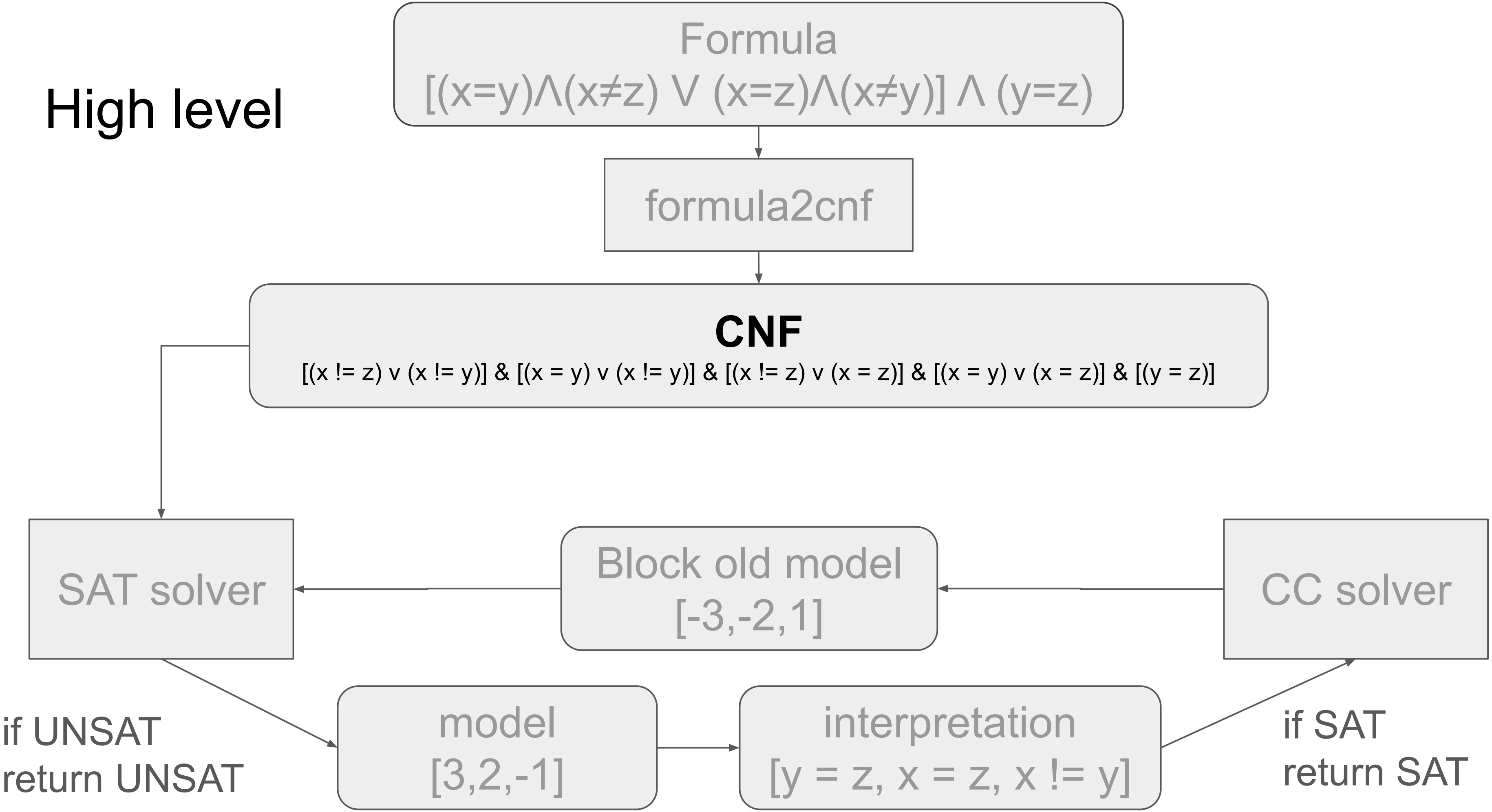
SimpleCongruenceClosure

- Input: Conjunctions of
  Term = Term | Term ≠ Term
  - Term is
    Var | F(Term)
- Output: SAT/UNSAT

```
CC_solv
All (di
    x !
    x =
    y =
Unique
    y
    z
    x
Disequa
    x !
Equival
    y
    z
    x
</CC so
Checkin                itself
testing
    aga
testing
    aga
Bottom!
```

```
<CC solver>
All (dis)equalities:
    F(x1) != F(x2)
    x1 = x2
Unique terms:
    x1
    x2
    F(x1)
    F(x2)
Disequalities
    F(x1) != F(x2)
Equivalence class 0:
    x1
    x2
</CC solver>
UNSAT
Should be UNSAT
```

High level



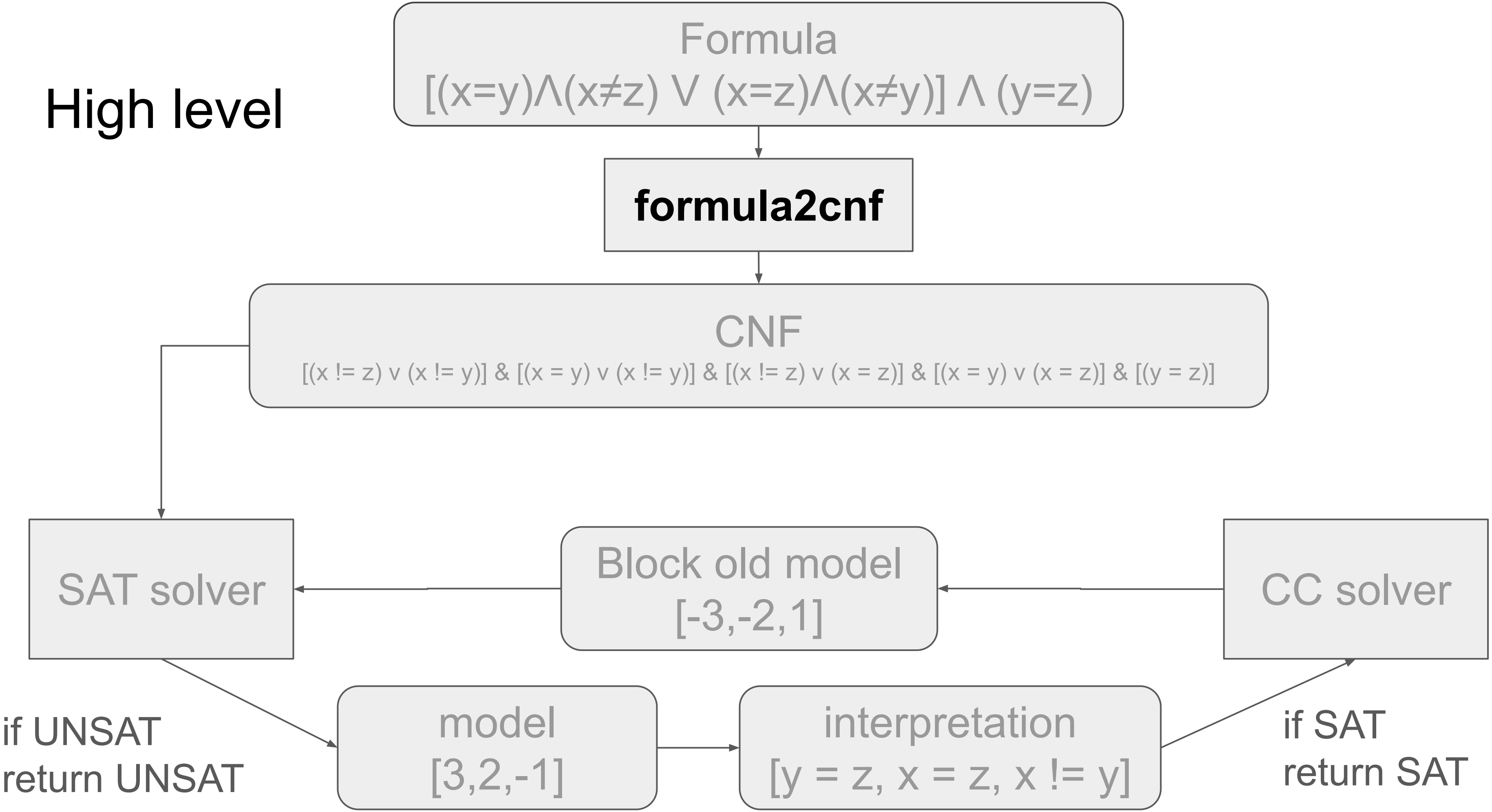
Formula
[(x=y)Λ(x≠z) V (x=z)Λ(x≠y)] Λ (y=z)
formula2cnf
CNF
[(x != z) v (x != y)] & [(x = y) v (x != y)] & [(x != z) v (x = z)] & [(x = y) v (x = z)] & [(y = z)]
SAT solver
Block old model
[-3,-2,1]
CC solver
if UNSAT
return UNSAT
model
[3,2,-1]
interpretation
[y = z, x = z, x != y]
if SAT
return SAT

CNF

- A class with a list of conjuncts
- Literals are also on the form
  Term=Term | Term≠Term
- and(cnf1, cnf2): trivial
- not(cnf): DeMorgan
- or(cnf1, cnf2): distribution

```
c1: [(a = a) v (b = b)] & [(c = c) v (d = d)]
c2: [(e = e) v (f = f)] & [(g = g) v (h = h)]
c1 or c2: [(e = e) v (f = f) v (a = a) v (b = b)] & [(g = g) v (h = h) v (a = a) v (b
= b)] & [(e = e) v (f = f) v (c = c) v (d = d)] & [(g = g) v (h = h) v (c = c) v (d =
d)]
```

High level

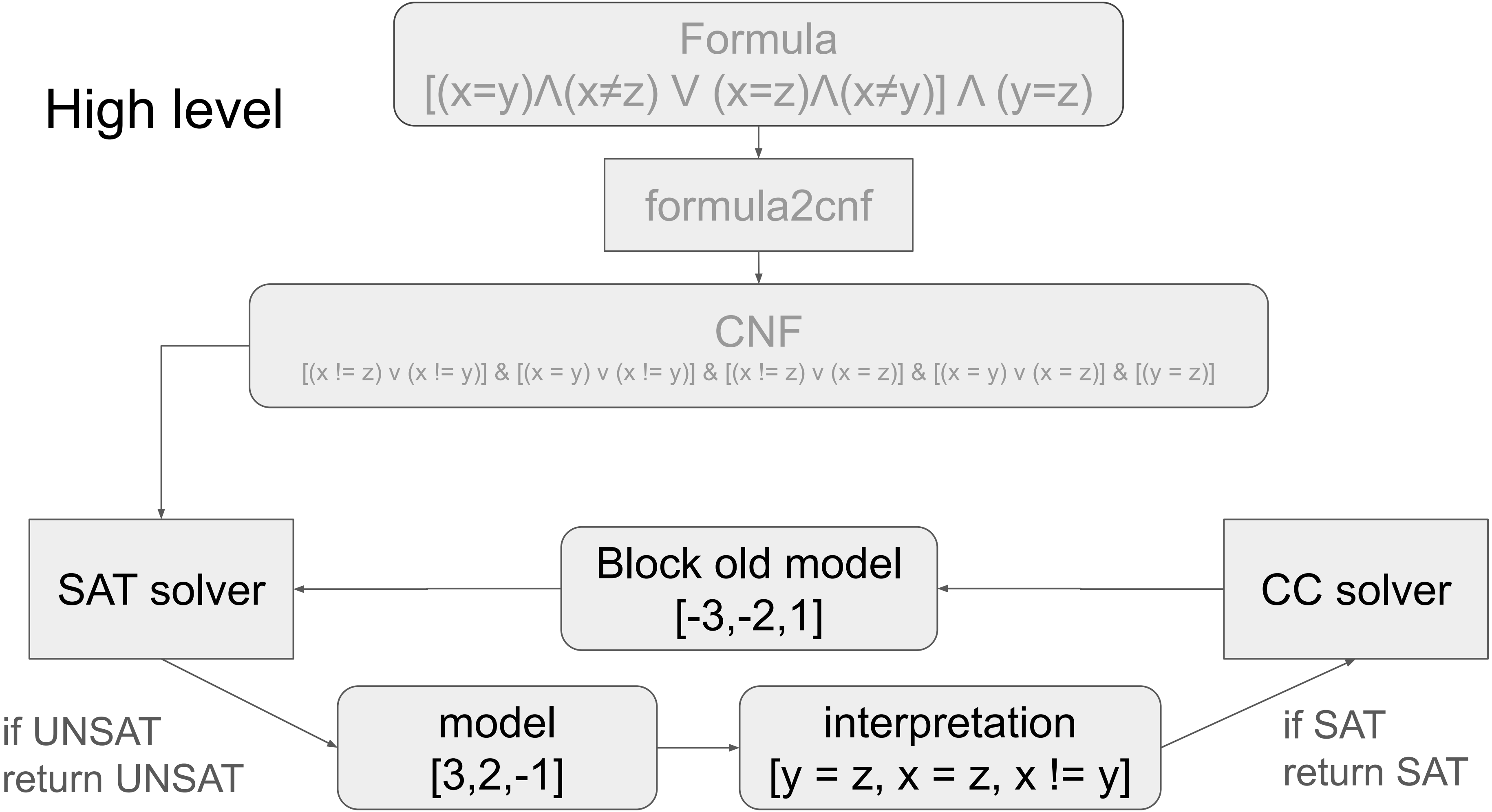


Formula
[(x=y)Λ(x≠z) V (x=z)Λ(x≠y)] Λ (y=z)
formula2cnf
CNF
[(x != z) v (x != y)] & [(x = y) v (x != y)] & [(x != z) v (x = z)] & [(x = y) v (x = z)] & [(y = z)]
SAT solver
Block old model
[-3,-2,1]
CC solver
if UNSAT
return UNSAT
model
[3,2,-1]
interpretation
[y = z, x = z, x != y]
if SAT
return SAT

formula2cnf

- Naive conversion to cnf is simple
  - Push all negations down to literals
  - Exponential blow-up
- Proper way: Tseitin algorithm
  - Introduce “extra” variables
  - Linear size
  - Satisfiability-preserving
- In the interest of time: Naive way

High level



Formula
[(x=y)Λ(x≠z) V (x=z)Λ(x≠y)] Λ (y=z)
formula2cnf
CNF
[(x != z) v (x != y)] & [(x = y) v (x != y)] & [(x != z) v (x = z)] & [(x = y) v (x = z)] & [(y = z)]
SAT solver
Block old model
[-3,-2,1]
CC solver
if UNSAT
return UNSAT
model
[3,2,-1]
interpretation
[y = z, x = z, x != y]
if SAT
return SAT

Solver loop

```python
# The SAT model is invalid.
# Negate it and add it as a clause to the SAT solver.
neg_model_clause = []
for literal in sat_model:
    neg_model_clause.append(literal * -1)
if verbose:
    print(f"neg_model_clause: {neg_model_clause}")
SAT_solver.add_clause(neg_model_clause)
```

A word about ordering

- I did not want to deal with symmetry etc: $a=b \Leftrightarrow b=a$
- Impose a total ordering for each class,
  enforced when the class is instantiated
- Atom(a,b) and Atom(b,a) will create the same thing: $a=b$

SMT things not covered

- A lot.
- Main SMT solvers: z3 and cvc5
  - Both have python wrappers that easy to use
- Quantifiers!
  - Pose a huge challenge for SMT solvers
- Other Theories
- Real-world examples