# SAT-based Analysis for C Programs

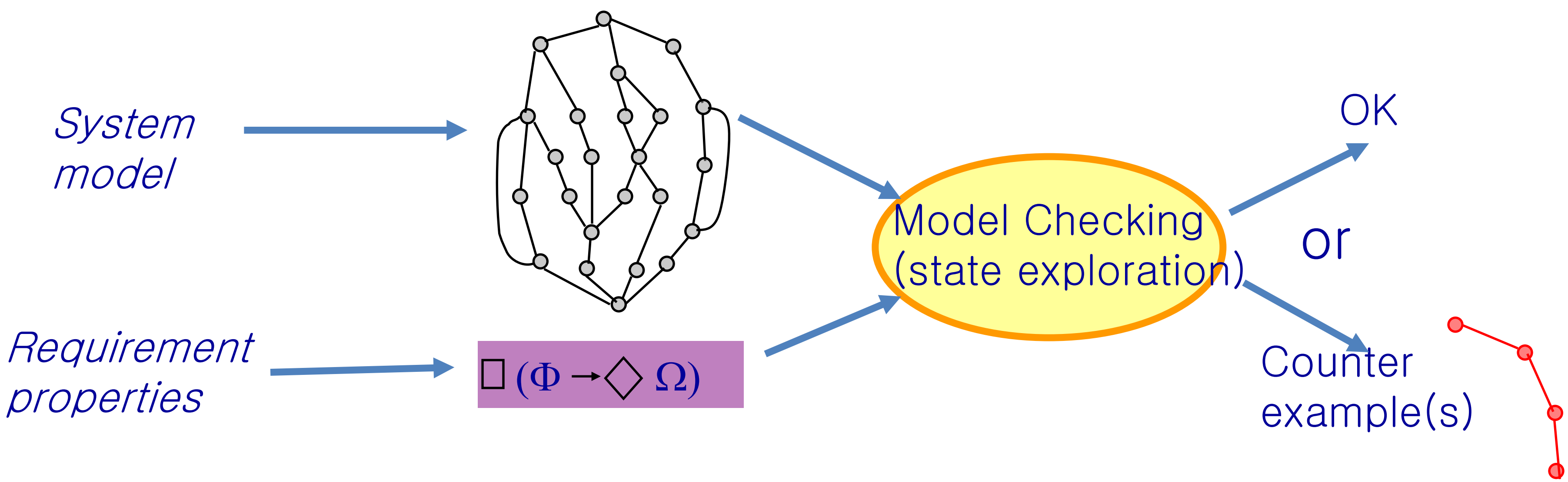
Moonzoo Kim
Provable Software Lab, KAIST

# Contents

- PART I: Model Checking Basics
- PART II: SAT-based Bounded Model Checking of C programs
- PART III: MiniSAT SAT Solver

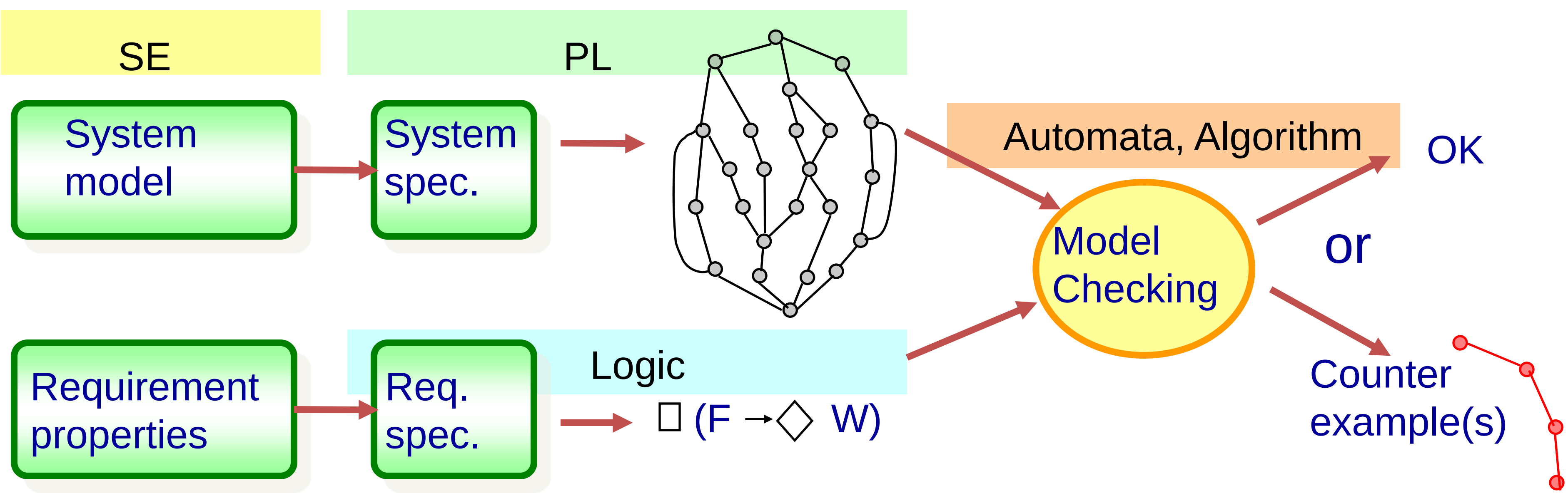
# Model Checking Basics

- Specify requirement properties and build a system model
- Generate possible states from the model and then check whether given requirement properties are satisfied within the state space



# Model Checking Basics (cont.)

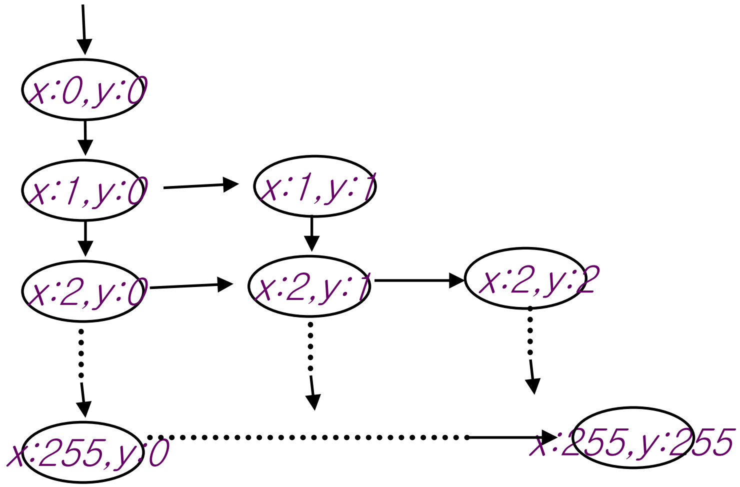
- Undergraduate foundational CS classes contribute this area
  - CS204 Discrete mathematics
  - CS300 Algorithm
  - CS320 Programming language
  - CS322 Automata and formal language
  - CS350 Introduction to software engineering
  - CS402 Introduction to computational logic



# An Example of Model Checking ½
## *(checking every possible execution path)*

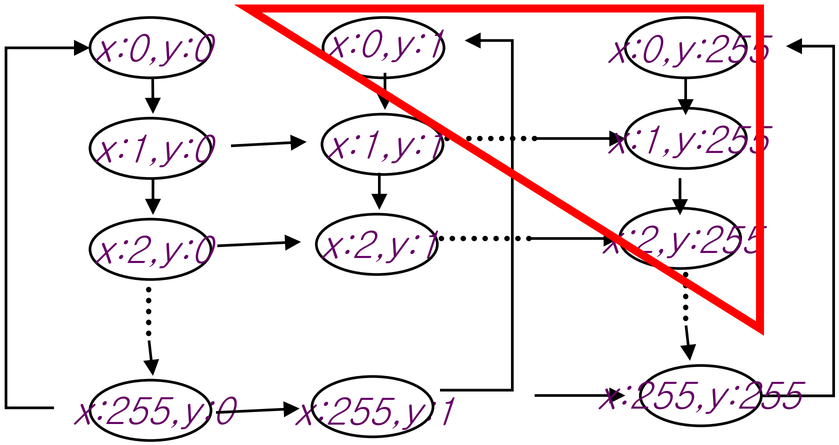
System Spec.

```
unsigned char x=0;
unsigned char y=0;

void proc_A()
  while(1)
    x++;
}

void proc_B(){
  while(1)
    if (x>y)
      y++;
}
```




Req. Spec

□ (x >= y)



# An Example of Model Checking 2/2

## (*checking every possible thread scheduling* )

```
char cnt=0,x=0,y=0,z=0;

void process() {
        char me = _pid +1; /* me is 1 or 2*/
again:
        x = me;
        If (y ==0 || y== me) ;
        else goto again;

        z =me;
        If (x == me) ;
        else goto again;

        y=me;
        If(z==me);
        else goto again;

        /* enter critical section */
        cnt++;
        assert( cnt ==1);
        cnt --;
        goto again;
}
```

*Software locks*

*Critical section*

*Mutual Exclusion Algorithm*

| *Process 0* | *Process 1* |
|---|---|
| | x = 2<br>y==0 \|\| y ==2<br>z = 2<br>x==2 |
| x = 1<br>y==0 \|\| y == 1 | |
| | y=2<br>(z==2)<br>cnt++ |
| z = 1<br>x==1<br>y = 1<br>z == 1<br>cnt++ | |

*Violation detected !!!*

*Counter Example*

# Motivation

- Data flow analysis: fastest & least precision
  - "May" analysis,
- Abstract interpretation: fast & medium precision
  - Over-approximation & under-approximation
- Model checking: slow & complete
  - Complet value analysis
  - No approximation

- Static analyzer & MC as a C debugger
  - Handling complex C structures such as pointer and array
    - DFA: might-be
    - AI: may-be
    - MC: can-be
    - SAT-based MC: (almost)complete

# Example. Sort (1/2)

| | | | |
|---|---|---|---|
| 9 | 14 | 2 | 255 |

- Suppose that we have an array of 4 elements each of which is 1 byte long
  - unsigned char a[4];
- We wants to verify sort.c works correctly
  - main() { sort(); assert(a[0]<= a[1]<= a[2]<=a[3]);}
- Explicit model checker (ex. Spin) requires at least $2^{32}$ bytes of memory
  - 4 bytes = 32 bits, No way…
- Symbolic model checker (ex. NuSMV) takes 200 megabytes in 400 sec

# Example. Sort (2/2)

```
#include <stdio.h>
#define N 5
int main(){
    int data[N], i, j, tmp;
    /* Assign random values to the array*/
    for (i=0; i<N; i++){
        data[i] = nondet_int();
    }
    /* It misses the last element, i.e., data[N-1]*/
    for (i=0; i<N-1; i++)
        for (j=i+1; j<N-1; j++)
            if (data[i] > data[j]){
                tmp = data[i];
                data[i] = data[j];
                data[j] = tmp;
            }
    /* Check the array is sorted */
    for (i=0; i<N-1; i++){
        assert(data[i] <= data[i+1]);
    }
}
```

- Total 6224 CNF clause with 19099 boolean propositional variables
- Theoretically, $2^{19099}$ ($2.35 \times 10^{5749}$) choices should be evaluated!!!

| SAT | VSIDS | Modified |
|---|---|---|
| Conflicts | 73 | 72 |
| Decisions | 2435 | 2367 |
| Time(sec) | 0.015 | 0.013 |

| UNSAT | VSIDS | Modified |
|---|---|---|
| Conflicts | 35067 | 30910 |
| Decisions | 161406 | 159978 |
| Time(sec) | 1.89 | 1.60 |

# PART I: SAT-based Bounded Model Checking

- Model Checking History
- SAT Basics
- Model Checking as a SAT problem

# Model Checking History

1981 Clarke / Emerson: CTL Model Checking
Sifakis / Quielle
1982 EMC: Explicit Model Checker
Clarke, Emerson, Sistla

**$10^5$**

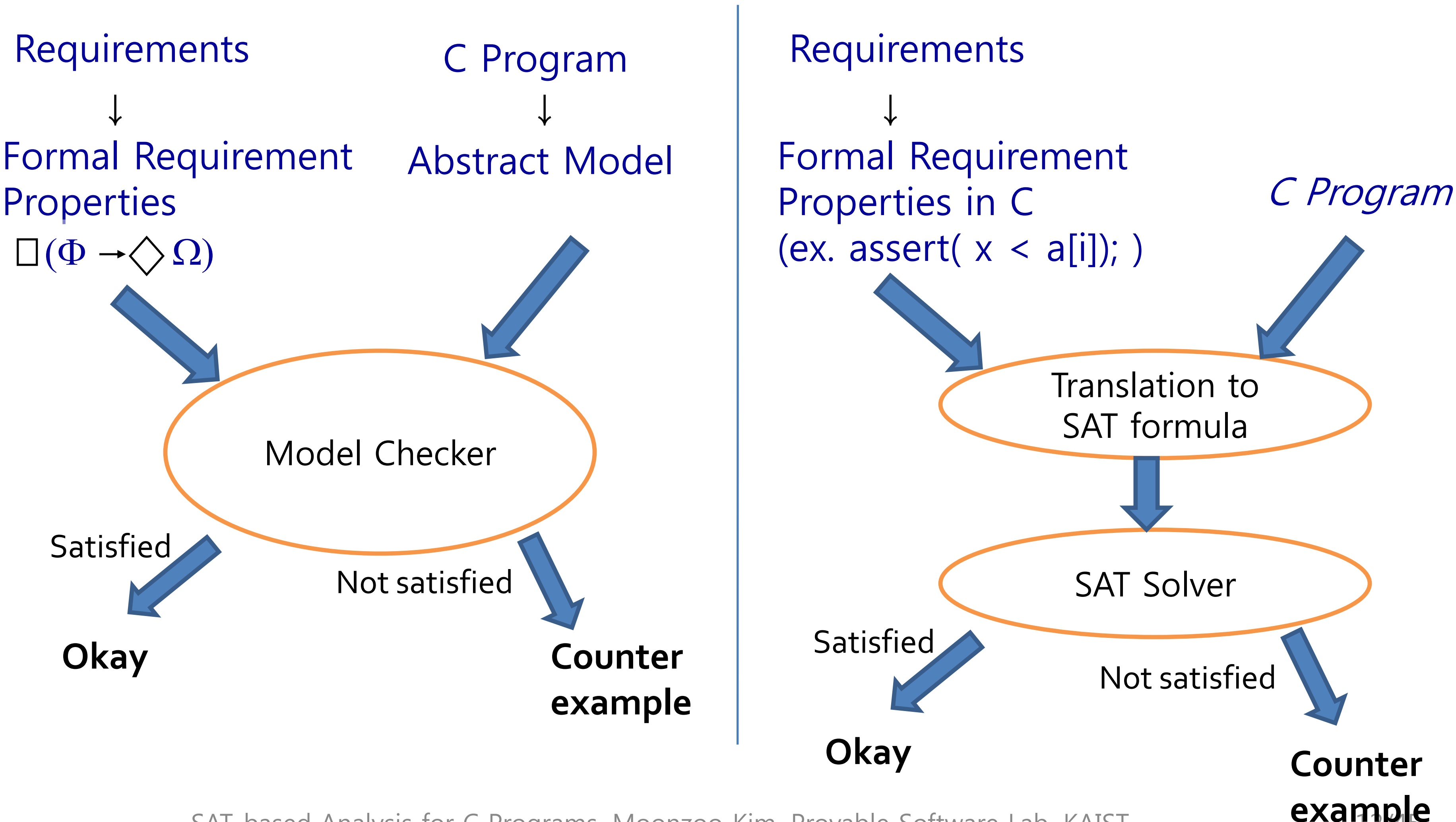
1990 Symbolic Model Checking
Burch, Clarke, Dill, McMillan
1992 SMV: Symbolic Model Verifier
McMillan

**$10^{100}$**

1998 Bounded Model Checking using SAT
Biere, Clarke, Zhu
2000 Counterexample-guided Abstraction Refinement
Clarke, Grumberg, Jha, Lu, Veith

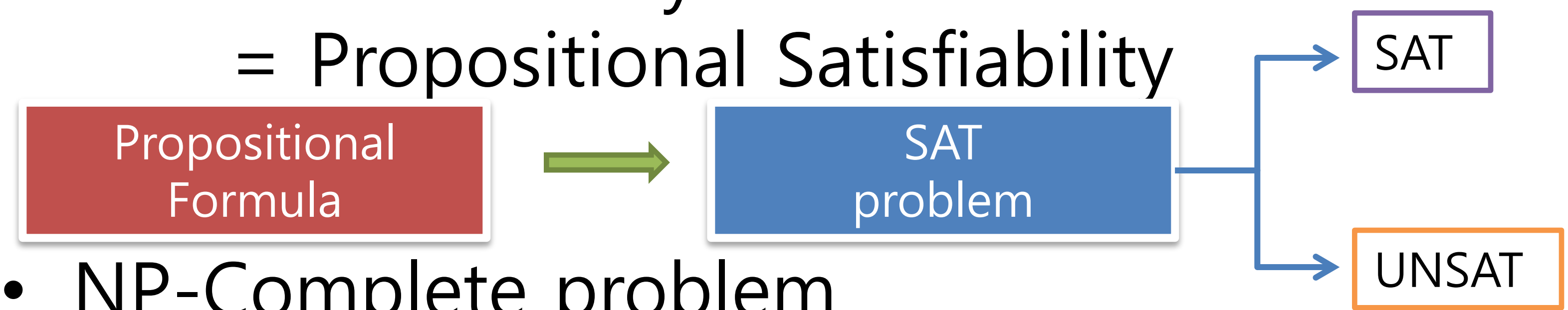
**$10^{1000}$**

# Overview of SAT-based Bounded Model Checking



# SAT Basics (1/2)

- SAT = Satisfiability
  = Propositional Satisfiability




- NP-Complete problem
  - We can use SAT solver for many NP-complete problems
    - Hamiltonian path
    - 3 coloring problem
    - Traveling sales man's problem
- Recent interest as a verification engine

# SAT Basics (2/2)

- A set of propositional variables and clauses involving variables
  - $(x_1 \vee x_2' \vee x_3) \wedge (x_2 \vee x_1' \vee x_4)$
  - $x_1$, $x_2$, $x_3$ and $x_4$ are variables (true or false)

- Literals: Variable and its negation
  - $x_1$ and $x_1'$

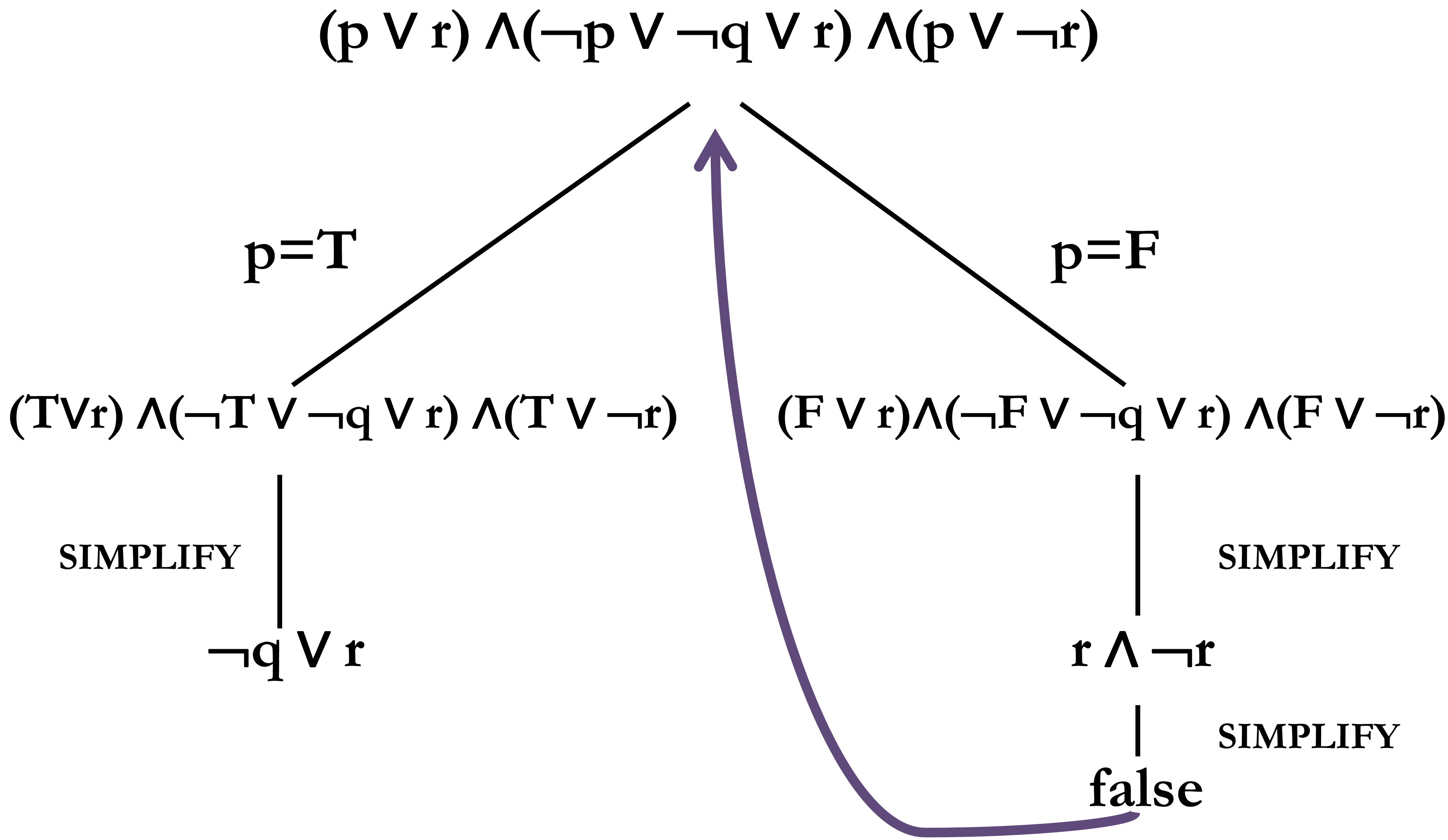
- A clause is satisfied if one of the literals is true
  - $x_1$=true satisfies clause 1
  - $x_1$=false satisfies clause 2

- Solution: An assignment that satisfies all clauses

# Basic SAT Solving Mechanism (1/2)

```
/* The Quest for Efficient Boolean Satisfiability Solvers
 *  by L.Zhang and S.Malik, Computer Aided Verification 2002 */
DPLL(a formula Á, assignment) {
    necessary = deduction(Á, assignment);
    new_asgnment = union(necessary, assignment);
    if (is_satisfied(Á, new_asgnment))
            return SATISFIABLE;
    else if (is_conflicting(Á, new_asgnmnt))
            return UNSATISFIABLE;
    var = choose_free_variable(Á, new_asgnmnt);
    asgn1 = union(new_asgnmnt, assign(var, 1));
    if (DPLL(Á, asgn1) == SATISFIABLE)
            return SATISFIABLE;
    else {
            asgn2 = union (new_asgnmnt, assign(var,0));
            return DPLL (Á, asgn2);
    }
}
```

# Basic SAT Solving Mechanism (2/2)



# Model Checking as a SAT problem (1/4)

- CBMC (C Bounded Model Checker, In CMU)
  - Handles function calls using inlining
  - Unwinds the loops a fixed number of times
  - Allows user input to be modeled using non-determinism
    - So that a program can be checked for a set of inputs rather than a single input
  - Allows specification of assertions which are checked using the bounded model checking

# Model Checking as a SAT problem (2/4)

- Unwinding Loop

Original code

```
x=0;
while (x < 2) {
  y=y+x;
  x++;
}
```

Unwinding the loop 3 times

```
x=0;
if (x < 2) {
  y=y+x;
  x++;
}
if (x < 2) {
  y=y+x;
  x++;
}
if (x < 2) {
  y=y+x;
  x++;
}
```

Unwinding assertion: →

```
assert (! (x < 2))
```

# Model Checking as a SAT problem (3/4)

- From C Code to SAT Formula

Original code

```
x=x+y;
if (x!=1)
  x=2;
else
  x++;
assert(x<=3);
```

Convert to static single assignment

$x_1=x_0+y_0;$
if $(x_1!=1)$
  $x_2=2;$
else
  $x_3=x_1+1;$
$x_4=(x_1!=1)?x_2:x_3;$
assert$(x_4<=3);$

Generate constraints

$C \equiv x_1=x_0+y_0 \land x_2=2 \land x_3=x_1+1 \land (x_1!=1 \land x_4=x_2 \lor x_1=1 \land x_4=x_3)$

$P \equiv x_4 <= 3$

Check if $C \land \neg P$ is satisfiable, if it is then the assertion is violated

$C \land \neg P$ is converted to Boolean logic using a bit vector representation for the integer variables $y_0, x_0, x_1, x_2, x_3, x_4$

# Model Checking as a SAT problem (4/4)

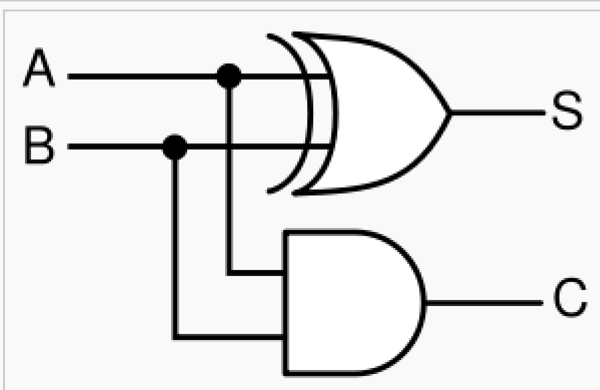
- Example of arithmetic encoding into pure propositional formula

Assume that x,y,z are three bits positive integers represented by propositions $x_0x_1x_2$, $y_0y_1y_2$, $z_0z_1z_2$

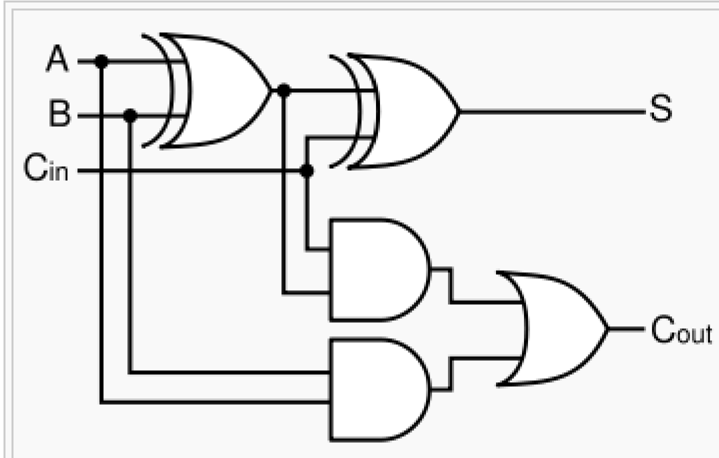
$C \equiv z=x+y \equiv \ (z_0\$(x_0\text{©}y_0)\text{©}(\ (x_1\text{Æ}y_1)\ \text{Ç}\ (((x_1\text{©}y_1)\text{Æ}(x_2\text{Æ}y_2)))$

$\text{Æ}\ (z_1\$(x_1\text{©}y_1)\text{©}(x_2\text{Æ}y_2))$

$\text{Æ}\ (z_2\$(x_2\text{©}y_2))$



Half adder circuit diagram



Full adder circuit diagram
**Inputs: {A, B, CarryIn} → Outputs: {Sum, CarryOut}**

# PART II: MiniSAT SAT Solver

- Overview
- Conflict Clause Analysis
- VSIDS Decision Heuristic

# SAT Solver History

- Started with DPLL (1962)
  - Able to solve 10-15 variable problems
- Satz (Chu Min Li, 1995)
  - Able to solve some 1000 variable problems
- Chaff (Malik et al., 2001)
  - Intelligently hacked DPLL , Won the 2004 competition
  - Able to solve some 10000 variable problems
- Current state-of-the-art
  - MiniSAT and SATELITEGTI (Chalmer's university, 2004-2006)
  - Jerusat and Haifasat (Intel Haifa, 2002)
  - Ace (UCLA, 2004-2006)

# Overview

- MiniSat is a fast SAT solver developed by Niklas Eén a nd Niklas Sörensson
  - MiniSat won all industrial categories in SAT 2005 competition
  - MiniSat won SAT-Race 2006
- MiniSat is simple and well-documented
  - Well-defined interface for general use
  - Helpful implementation documents and comments
  - Minimal but efficient heuristic

# Overview

- Unit clause is a clause in which all but one of literals is assigned to False
- Unit literal is the unassigned literal in a unit clause

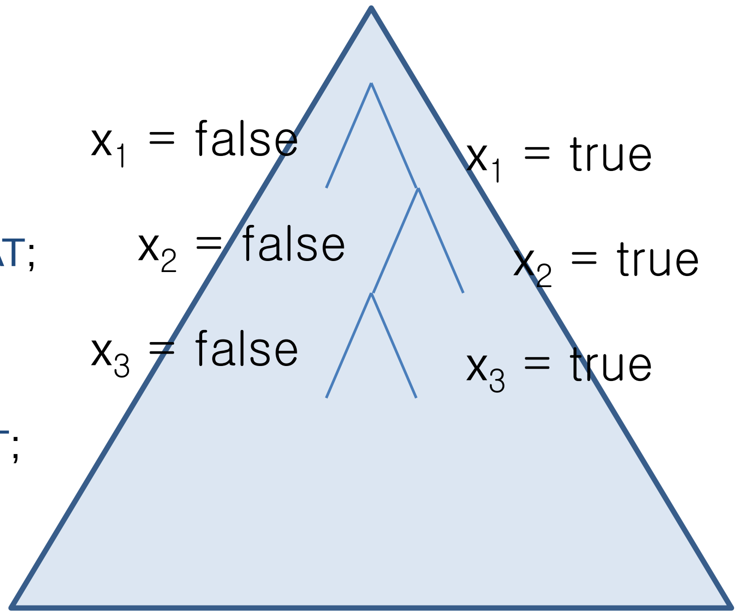
$$\begin{array}{l} \ldots\ldots \\ (x_0) \wedge \\ (-x_0 \vee x_1) \wedge \\ (-x_2 \vee -x_3 \vee -x_4) \wedge \\ \ldots\ldots \end{array}$$

  - $(x_0)$ is a unit clause and '$x_0$' is a unit literal
  - $(-x_0 \vee x_1)$ is a unit clause since $x_0$ has to be True
  - $(-x_2 \vee -x_3 \vee -x_4)$ can be a unit clause if the current assignment is that $x_3$ and $x_4$ are True
- Boolean Constrain Propagation(BCP) is the process of assigning the True value to all unit literals

# Overview

/* overall structure of Minisat solve procedure */

```
Solve(){
    while(true){
        boolean_constraint_propagation();
        if(no_conflict){
            if(no_unassigned_variable) return SAT;
            decision_level++;
            make_decision();
        }else{
            if (no_dicisions_made) return UNSAT;
            analyze_conflict();
            undo_assignments();
            add_conflict_clause();
        }
    }
}
```



# Conflict Clause Analysis

- A conflict happens when one clause is falsified by unit propagation

**Assume $x_4$ is False**
**$(x_1 \vee x_4) \wedge$**
**$(-x_1 \vee x_2) \wedge$**
**$(-x_2 \vee x_3) \wedge$**
**$(-x_3 \vee -x_2 \vee -x_1)$ Falsified!**

- Analyze the conflicting clause to infer a clause
  - $(-x_3 \vee -x_2 \vee -x_1)$ is conflicting clause
- The inferred clause is a new knowledge
  - A new learnt clause is added to constraints

# Conflict Clause Analysis

- Learnt clauses are inferred by conflict analysis

$(x_1 \vee x_4) \wedge$
$(-x_1 \vee x_2) \wedge$
$(-x_2 \vee x_3) \wedge$
$(-x_3 \vee -x_2 \vee -x_1) \wedge$
**$(x_4)$ learnt clause**

- They help prune future parts of the search space
  - Assigning False to $x_4$ is the casual of conflict
  - Adding (x4) to constraints prohibit conflict from $-x_4$
- Learnt clauses actually drive backtracking

# Conflict Clause Analysis

```
/* conflict analysis algorithm */
Analyze_conflict(){
    cl = find_confclicting_clause();
    /* Loop until cl is falsified and one literal whose value is determined in current
    decision level is remained */
    While(!stop_criterion_met(cl)){
        lit = choose_literal(cl); /* select the last propagated literal */
        Var = variable_of_literal(lit);
        ante = antecedent(var);
        cl = resolve(cl, ante, var);
    }
    add_clause_to_database(cl);
    /* backtrack level is the lowest decision level for which the learnt clause is
    unit clause */
    back_dl = clause_asserting_level(cl);
    return back_dl;
}
```

Algorithm from Lintao Zhang and Sharad malik
"The Quest for Efficient Boolean Satisfiability Solvers"

# Conflict Clause Analysis

- Example of conflict clause analysis

$(-f \lor e) \land$
$(-g \lor f) \land$
$(b \lor a \lor e) \land$
$(c \lor e \lor f \lor -b) \land$
$(d \lor -b \lor h) \land$
$(-b \lor -c \lor -d) \land$
$(c \lor d)$

Satisfiable?

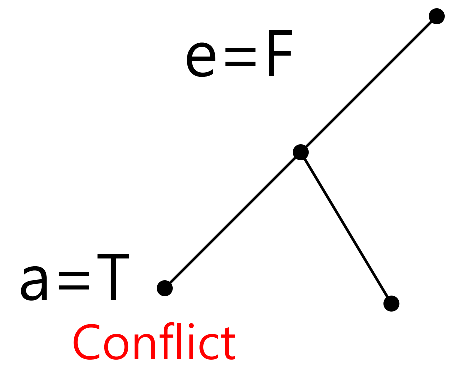
Unsatisfiable?

# Conflict Clause Analysis

| Assignments | antecedent |
|---|---|
| e=F | assumption |
| f=F | -f∨e |
| g=F | -g∨f |
| h=F | -h∨g |
| a=F | assumption |
| b=T | b∨a∨e |
| c=T | c∨e∨f∨-b |
| d=T | d∨-b∨h |

DLevel=1 (e=F, f=F, g=F, h=F)

DLevel=2 (a=F, b=T, c=T, d=T)

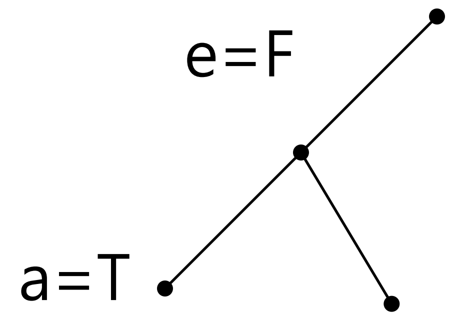



-b∨-c∨-d

Example slides are from CMU 15-414 course ppt

# Conflict Clause Analysis

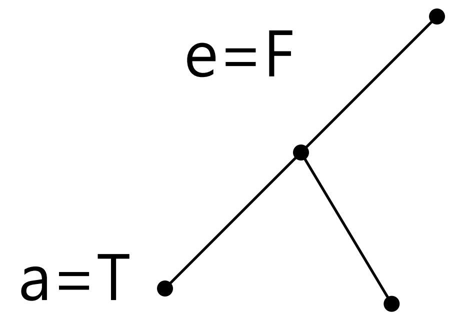
| Assignments | | antecedent |
|---|---|---|
| e=F | | assumption |
| f=F | DLevel=1 | -f∨e |
| g=F | | -g∨f |
| h=F | | -h∨g |
| a=F | | assumption |
| b=T | DLevel=2 | b∨a∨e |
| c=T | | c∨e∨f∨-b |
| d=T | | d∨-b∨h |




-b∨-c∨-d

# Conflict Clause Analysis

| Assignments | antecedent |
|---|---|
| e=F (DLevel=1) | assumption |
| f=F | -f∨e |
| g=F | -g∨f |
| h=F | -h∨g |
| a=F (DLevel=2) | assumption |
| b=T | b∨a∨e |
| c=T | c∨e∨f∨-b |
| d=T | d∨-b∨h |




-b∨-c∨-d

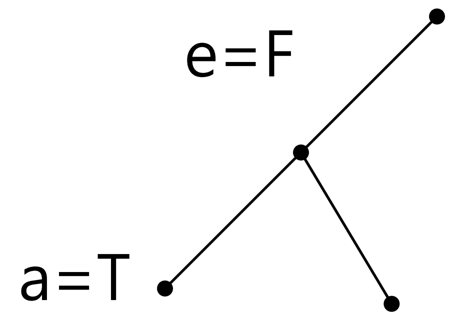
-b∨-c∨h

# Conflict Clause Analysis

| Assignments | antecedent |
|---|---|
| e=F | assumption |
| f=F | -f∨e |
| g=F | -g∨f |
| h=F | -h∨g |
| a=F | assumption |
| b=T | b∨a∨e |
| c=T | c∨e∨f∨-b |
| d=T | d∨-b∨h |

DLevel=1 (e=F, f=F, g=F, h=F)

DLevel=2 (a=F, b=T, c=T, d=T)

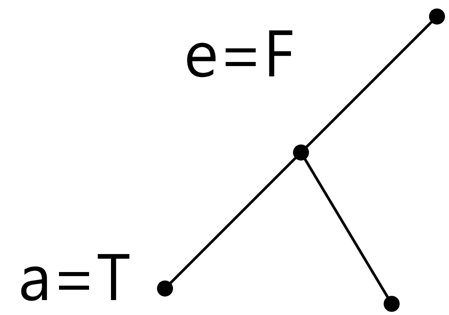



-b∨-c∨-d

-b∨-c∨h

# Conflict Clause Analysis

| Assignments | | antecedent |
|---|---|---|
| e=F | DLevel=1 | assumption |
| f=F | | -f∨e |
| g=F | | -g∨f |
| h=F | | -h∨g |
| a=F | DLevel=2 | assumption |
| b=T | | b∨a∨e |
| c=T | | c∨e∨f∨-b |
| d=T | | d∨-b∨h |




-b∨-c∨-d

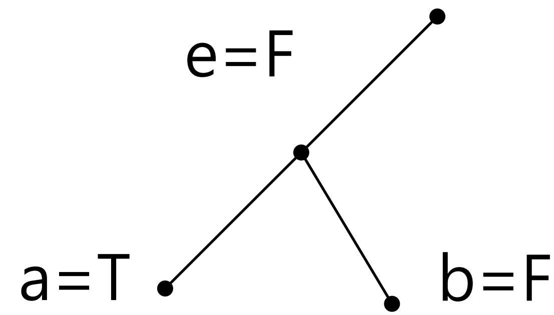
-b∨-c∨h

-b∨e∨f∨h learnt clause

# Conflict Clause Analysis

| Assignments | antecedent |
|---|---|
| e=F | assumption |
| f=F | -f∨e |
| g=F | -g∨f |
| h=F | -h∨g |
| b=F | -b∨e∨f∨h |
| … | … |
| | |
| | |

DLevel=1




-b∨-c∨-d

-b∨-c∨h

-b∨e∨f∨h

# VSIDS Decision Heuristic

- Variable State Independent Decaying Sum(VSIDS)
  - decision heuristic to determine what variable will be assigned next
  - decision is independent from the current assignment of each variable

- VSIDS makes decisions based on activity
  - Activity is a literal occurrence count with higher weight on the more recently added clauses
  - MiniSAT does not consider any polarity in VSIDS
    - Each variable, not literal has score

activity description from Lintao Zhang and Sharad malik
"The Quest for Efficient Boolean Satisfiability Solvers"

# VSIDS Decision Heuristic

- Initially, the score for each variable is 0
- First make a decision e = False
  - The order between same score is unspecified.
  - MiniSAT always assigns False to variables.

**Initial constraints**
**(-f∨e) ∧**
**(-g∨f) ∧**
**(b∨a∨e) ∧**
**(c∨e∨f∨-b) ∧**
**(d∨-b∨h) ∧**
**(-b∨-c∨-d) ∧**
**(c∨d)**

| Variable | Score | Value |
|---|---|---|
| a | **0** | |
| b | **0** | |
| c | **0** | |
| d | **0** | |
| e | **0** | **F** |
| f | **0** | |
| g | **0** | |
| h | **0** | |

# VSIDS Decision Heuristic

- f, g, h are False after BCP

**(-f∨e) ∧**
**(-g∨f) ∧**
**(b∨a∨e) ∧**
**(c∨e∨f∨-b) ∧**
**(d∨-b∨h) ∧**
**(-b∨-c∨-d) ∧**
**(c∨d)**

| Variable | Score | Value |
|---|---|---|
| a | **0** | |
| b | **0** | |
| c | **0** | |
| d | **0** | |
| e | **0** | **F** |
| f | **0** | **F** |
| g | **0** | **F** |
| h | **0** | **F** |

# VSIDS Decision Heuristic

- a is next decision variable

(-f∨e) ∧
(-g∨f) ∧
**(b∨a**∨e**)** ∧
**(c**∨e∨f**∨-b)** ∧
**(d∨-b**∨h**)** ∧
**(-b∨-c∨-d)** ∧
**(c∨d)**

| Variable | Score | Value |
|---|---|---|
| a | **0** | **F** |
| b | **0** | |
| c | **0** | |
| d | **0** | |
| e | **0** | **F** |
| f | **0** | **F** |
| g | **0** | **F** |
| h | **0** | **F** |

# VSIDS Decision Heuristic

- b, c are True after BCP
- Conflict occurs on variable d
  - Start conflict analysis

(-f∨e) ∧
(-g∨f) ∧
(b∨a∨e) ∧
(c∨e∨f∨-b) ∧
(d∨-b∨h) ∧
**(-b∨-c∨-d)** ∧
(c∨d)

| Variable | Score | Value |
|---|---|---|
| a | **0** | **F** |
| b | **0** | **T** |
| c | **0** | **T** |
| d | **0** | **T** |
| e | **0** | **F** |
| f | **0** | **F** |
| g | **0** | **F** |
| h | **0** | **F** |

# VSIDS Decision Heuristic

- The score of variable in resolvents is increased by 1
- If a variable appears in resolvents two or mores increase the score just once

**(-f∨e) ∧**
**(-g∨f) ∧**
**(b∨a∨e) ∧**
**(c∨e∨f∨-b) ∧**
**(d∨-b∨h) ∧**
**(-b∨-c∨-d) ∧**
**(c∨d)**

**Resolvent on d**
**-b∨-c∨h**

| Variable | Score | Value |
| --- | --- | --- |
| a | **0** | **F** |
| b | **1** | **T** |
| c | **1** | **T** |
| d | **0** | **T** |
| e | **0** | **F** |
| f | **0** | **F** |
| g | **0** | **F** |
| h | **1** | **F** |

# VSIDS Decision Heuristic

- The end of conflict analysis
- The scores are decaying 5% for next scoring

**(-f∨e) ∧**
**(-g∨f) ∧**
**(b∨a∨e) ∧**
**(c∨e∨f∨-b) ∧**
**(d∨-b∨h) ∧**
**(-b∨-c∨-d) ∧**
**(c∨d)**

**Resolvents**
**-b∨-c∨h**
**-b∨e∨f∨h ← learnt clause**

| Variable | Score | Value |
|---|---|---|
| a | **0** | **F** |
| b | **0.95** | **T** |
| c | **0.95** | **T** |
| d | **0** | **T** |
| e | **0.95** | **F** |
| f | **0.95** | **F** |
| g | **0** | **F** |
| h | **0.95** | **F** |

# VSIDS Decision Heuristic

- b is now False and a is True after BCP
- Next decision variable is c with 0.95 score

(-f∨e) ∧
(-g∨f) ∧
(b∨a∨e) ∧
(c∨e∨f∨-b) ∧
(d∨-b∨h) ∧
(-b∨-c∨-d) ∧
**(c∨d) ∧**
**Learnt clause** (-b∨e∨f∨h )

| Variable | Score | Value |
|---|---|---|
| a | **0** | **T** |
| b | **0.95** | **F** |
| c | **0.95** | |
| d | **0** | |
| e | **0.95** | **F** |
| f | **0.95** | **F** |
| g | **0** | **F** |
| h | **0.95** | **F** |

# VSIDS Decision Heuristic

- Finally we find a model

(-f∨e) ∧
(-g∨f) ∧
(b∨a∨e) ∧
(c∨e∨f∨-b) ∧
(d∨-b∨h) ∧
(-b∨-c∨-d) ∧
**(c∨d) ∧**
**Learnt clause** (-b∨e∨f∨h )

| Variable | Score | Value |
|---|---|---|
| a | **0** | **T** |
| b | **0.95** | **F** |
| c | **0.95** | **F** |
| d | **0** | **T** |
| e | **0.95** | **F** |
| f | **0.95** | **F** |
| g | **0** | **F** |
| h | **0.95** | **F** |

# Basic References


- The Quest for Efficient boolean Satisfiability Solvers

  L. Zhang and S. Malik

  *Computer Aided Verification, Denmark, July 2002 (LNCS 2404)*
- A tool for checking ANSI-C programs

  E. Clarke, D. Kroening and F. Lerda

  *Tools and Algorithms for the Construction and Analysis of Systems, Spain, 2004 (LNCS 2988)*
- Backdoors To Typical Case Complexity.

  R. Williams, C. Gomes, and B. Selman.
  *International Joint Conference on Artificial Intelligence*