CSC2/455 Software Analysis and Improvement
An Introduction to SAT/SMT Solvers

Sreepathi Pai
April 10, 2023
URCS
Outline

Introduction

SAT Solving

SMT Solvers

Applications to Program Analysis
Outline

Introduction

SAT Solving

SMT Solvers

Applications to Program Analysis
So far:

- Iterative Data Flow Analysis
- Type Analysis
- Region Analysis
- Abstract Interpretation

Today:

- Satisfiability (SAT) Solvers
- Satisfiability Modulo Theories (SMT) Solvers

Next:

- Model Checking
- Symbolic Execution
- Hoare Logic
The Satisfiability (SAT) Problem

Given a formula in propositional logic (variables, true, false, \( \wedge, \lor, \neg \), and parentheses), is there an assignment to variables that makes the formula true?

- \((A \lor B \lor C) \land (\neg A \lor B)\) (conjunctive normal form, CNF)
- \((A \land B \land C) \lor (\neg A \land B)\) (disjunctive normal form, DNF)
- \(A \land \neg A\) (CNF)
Solutions

- \((A \lor B \lor C) \land (\neg A \lor B)\)
  - \(B = \text{true}\) is required in any satisfying assignment \((A\text{ and }C\text{ don't matter})\)

- \((A \land B \land C) \lor (\neg A \land B)\)
  - \(A, B, C\) all true is one satisfying assignment
  - \(A = \text{false}\) and \(B = \text{true}\) is another satisfying assignment

- \(A \land \neg A\) is obviously unsatisfiable
3-SAT is NP-Complete

- If the maximum number of variables in a clause of a CNF formula is $k$, we call that problem $k$-SAT
- 2-SAT is solvable in polynomial time
- 3-SAT is NP-complete
  - In worst case, must explore every possible assignment of values to each variable
There are many good SAT solvers now available
- Based on the Davis–Putnam-Logemann-Loveland (DPLL) algorithm
- Often enhanced with Conflict-Driven Clause Learning (CDCL)
- SAT is decidable, if untractable

Intractability not a hindrance usually
- Can scale to very large problems
- Millions of clauses
- See: The International SAT Competition

Applied to many hardware and software verification problems
SAT solvers return:
- SAT: if a satisfying assignment is found (and the values that satisfy the proposition)
- UNSAT: if no satisfying assignment exists
Prove \( \neg(A \land B) = (\neg A \lor \neg B) \)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th>P=\neg(A \land B)</th>
<th>\neg A</th>
<th>\neg B</th>
<th>Q=\neg A \lor \neg B</th>
<th>P \iff Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
</tbody>
</table>

The statement \( \neg(A \land B) = (\neg A \lor \neg B) \) is valid, it is true for all values of \( A \) and \( B \).
Proof using a SAT solver

- \( \neg(A \land B) \iff (\neg A \lor \neg B) \)
- \( (\neg A \lor \neg B) \iff \neg(A \land B) \)
- Recall that \( P \implies Q \) can be written as \( \neg P \lor Q \)
  - \( \neg\neg(A \land B) \lor (\neg A \lor \neg B) \)
  - \( \neg(\neg A \lor \neg B) \lor \neg(A \land B) \)
- So we have:
  - \( R = (A \land B) \lor (\neg A \lor \neg B) \)
  - \( S = \neg(\neg A \lor \neg B) \lor \neg(A \land B) \)
- For the proof, we need both \( R \) and \( S \) to be valid
  - If \( R \) is valid, what can we say about the satisfiability of \( \neg R \)?
The Satisfiability of $\neg R$

<table>
<thead>
<tr>
<th>$A$</th>
<th>$B$</th>
<th>$P = A \land B$</th>
<th>$\neg A$</th>
<th>$\neg B$</th>
<th>$Q = \neg A \lor \neg B$</th>
<th>$P \lor Q$</th>
<th>$\neg (P \lor Q)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
</tbody>
</table>

If $R$ is valid, then $\neg R$ is unsatisfiable!

- To prove a statement using a SAT solver:
  - Convert the statement to propositional logic
  - Negate it, and check for unsatisfiability

- Interesting corollary:
  - If the formula is SAT (implying the statement is false), the values that satisfy the statement are counter-examples
More details on SAT Solving (Book)

- Volume 4 Facsicle 6 of *The Art of Computer Programming*
More details on SAT/SMT Solving (Papers)

- Weissenbacher, Subramanyan, and Malik, Boolean Satisfiability Solvers: Techniques and Extensions
- Barrett, Sebastinani, Sheshia and Tinelli, Satisfiability Modulo Theories
Propositional logic can be extended with quantifiers
- \( \exists \), existential quantifier
- \( \forall \), universal quantifier
- This is First-order Logic (FOL)
- FOL is undecidable in general

Both propositional logic and first-order logic are still boolean

Can be extended with theories:
- Arithmetic: adds numbers, operators +, −, \( \times \), associativity, commutativity, etc.
- Functions: adds \( f(x) \)
- Bitvectors: model variables containing \( n \) bits (where \( n > 1 \))
Satisfiability Modulo Theories (SMT)

- A SMT solver checks for satisfiability *in a theory*
  - Can think of statements as propositional logic + theory
- Allows construction of "richer" statements
  - Can formulate propositions over integers, reals, etc.
  - Can use operators like $+, -, \times$, etc.
- Example: $\forall x,y \ x > y \implies x + 1 > y + 1$
  - True over integers ($\mathbb{Z}$)
  - False over machine integers/bitvectors
Some theories are decidable
  - Presburger arithmetic
Most theories are undecidable
However, some theories undecidable in general are decidable over quantifier-free fragments
So, results of a SMT solver can be:
  - SAT
  - UNSAT
  - Don’t know (or infinite loop)
Some SMT solvers

- Microsoft Z3
  - Used to be available online at rise4fun/Z3
  - Available in most Linux distributions
- CVC5
- Yices
- Many more...
The SMT-LIB language

- Common input/output language for most SMT solvers
  - Some solvers support their own language as well
- Lisp-like
- Documented at smtlib.org
- Allows easy switching between solvers
  - We will use Z3 for the most part
Let’s encode $\neg R$ from the previous example:

```lisp
(declare-fun B () Bool)
(declare-fun A () Bool)
(assert (not (or (and A B) (not A) (not B))))
(check-sat)
```

And we run it:

```
$ z3 p1.smt
unsat
```
#!/usr/bin/env python3

from z3 import *

s = Solver()
A, B = Bools('A B')

R = Or(And(A, B), Or(Not(A), Not(B)))
notR = Not(R)

s.add(notR)
print(s.check())

print(s.sexpr())  # prints the SMTLIB code

See: Bjørner et al., Programming Z3, for a nice introduction to programming Z3 using Python.
The Constraint Satisfaction Problem seeks to find an assignment of values to variables subject to constraints:

- Each variable has a domain of values
- Pick a variable, assign it a value, subject to constraints
- If all variables can be assigned values, SAT else backtrack

For SAT in propositional logic:

- Two values, True and False
- Constraint: formula must evaluate to true
Other Problems

- Dennis Yurichev’s free book “SAT/SMT by Example” is a wonderful collection of examples
  - Minesweeper
  - Sudoku
  - Test case generation, etc.
Outline

Introduction

SAT Solving

SMT Solvers

Applications to Program Analysis
Express program behaviour in some logic
Express program property in that logic
Check if the property holds (i.e. is valid)
```c
int min(int a, int b) {
    if(a < b)
        return a;
    else
        return a;
}

int x, y, r;

r = min(x, y);

assert(r == x || r == y);
assert(r <= x && r <= y);
```

- These assertions test that `min` always returns the minimum of `x` and `y`.
- But `assert` executes at runtime.
- We will seek to prove statically:
  - $\forall_{x,y} (\min(x, y) = x \lor \min(x, y) = y) \land (\min(x, y) \leq x \land \min(x, y) \leq y)$
  - over all program paths.
from z3 import *

s = Solver()
a, b, ret = Ints('a b ret')
ret = If(a < b, a, a)
#ret = If(a < b, a, b) # correct
cond = And(Or(ret == a, ret == b), And(ret <= a, ret <= b))
 s.add(Not(cond))
print(s.sexpr())

if s.check() == sat:
    print("Incorrect. Counterexample: ", s.model())
else:
    print("Correct")

Output:

...  
Incorrect. Counterexample:  [b = 0, a = 1]
• If $A$ is a source program and $B$ is the compiled version, we would like to prove that $A = B$
  - This is called *translation validation*
  - What I’ve been doing for your submissions

• Undecidable, in general

• Not interesting only to compiler writers
  - If you take a piece of code and refactor it, did you break anything?
The test case (1, 3) is not sufficient to exercise all paths in the program
  - And it misses the bug!

Can we find test cases to exercise all paths in the program?
SMT solvers are a marvellous piece of technology

Learn to use one!