SAT and SMT

Murphy Berzish

Overview

- Boolean Satisfiability (SAT) problem
- SAT solvers: basic algorithms, enhancements
- Limitations of SAT
- SMT solvers: overview
- Examples of SMT solvers (implementations)
- Practical applications of SMT

Acknowledgements

- Some ideas and definitions from ECE750T28 (Computer-Aided Reasoning for SE) notes
  - https://ece.uwaterloo.ca/~vganesh/TEACHING/W2015/ECE750-T28/index.html
- An example borrowed from another talk: “SAT Solving, SMT Solving and Program Verification”
  - http://www.win.tue.nl/mdseminar/pres/zantema-17-02-11.pdf
  - but the example was incorrect in that talk! I have fixed it
- An exampled borrowed from a dReal benchmark
  - dreal.github.io
- An example borrowed from “Reverse Engineering for Beginners” by Dennis Yurichev
  - beginners.re

Introduction to Logic

- Logic is fundamental to computer science
  - constraint satisfaction problems
  - compilers: type-checking
  - hardware verification
  - software verification
- Comes in many forms:
  - propositional logic
  - first-order logic
  - higher-order logic

Propositional Logic

- Informally, “Boolean expressions”
- Simplest terms: “atoms”
  - truth symbols (1 = true, 0 = false)
  - variables (p, q, r, p_1, q_1, ...)
- Next level up: “literals”
  - either an atom (A) or its negation (!A)
- Finally: “formulas”
  - either a literal (L) or an application of a logical connective to some formulas
  - connectives: !F (negation), F_1 & F_2 (conjunction), F_1 | F_2 (disjunction), F_1 -> F_2 (implication), F_1 <-> F_2 (if and only if)

What Does “Solve” Mean?

- Make an “interpretation” (assign either 1 or 0 to every variable in the formula)
- Substitute assignments for variables
- Evaluate each expression
  - 0 & 1 = 0; 0 | 1 = 1; 1 -> 1 = 1...
- Under an interpretation, every propositional formula evaluates to either 1 (true) or 0 (false)

What Does “Solve” Mean?

- A formula F is **satisfiable** iff there exists an interpretation such that F is true.
- If no such interpretation exists, F is **unsatisfiable**.
- If *every possible* interpretation makes F true, then F is **valid**.
  - Exercise: prove duality between satisfiability and validity, i.e. *“F is valid iff !F is unsatisfiable*”.

What Does “Solve” Mean?

- We can now define the **SAT problem**:
  - Given a Boolean (propositional) formula F, decide whether F is satisfiable.
- Sometimes we know the answer and want the interpretation; sometimes we just want to know whether a solution exists
- The job of a SAT solver is to find a satisfying interpretation, or discover that none exist

Easy Way Out

- Why do we need special solvers?
- Try brute force!
- For a formula with N variables, how many interpretations?
  - Each variable can be either 0 or 1, so 2 possibilities
  - For N variables, 2^N interpretations to check
- Oops, this is exponential in the worst case.

Not So Fast

- In fact, SAT is NP-complete
- This means that all of our current algorithms to solve SAT are worst-case exponential
- How do we solve these things at all?
- SAT solvers are very interesting
  - they're still exponential-time in the worst case
  - but for many “practical” problems they are efficient!

How Do You Solve Sudoku?

- A lot of people have a very similar strategy:
  - Figure out which squares have only one possible value, and write those values there
  - Then repeat this until you can't do it any more
  - Now guess a (possible) value for some square
  - Repeat this until you solve the puzzle
  - If you get stuck, go back, make a different guess

How Do You Solve Sudoku?

- This can be expressed as an algorithm:
  - Davis-Putnam-Logemann-Loveland (DPLL)
- DPLL is a search algorithm for solving SAT!
- First incarnation as Davis-Putnam algorithm in 1962; refined to become DPLL

The DPLL Algorithm

- Unit resolution
  - deduce new information
  - a restricted form of a general procedure called “resolution”
- Given two clauses:
  - C_1 : p (a single literal, called a **unit clause**)
  - C_2 : $(L_1 \mid L_2 \mid ... \mid !p \mid ... \mid L_n)$
- Remove !p term from C_2 and rewrite to obtain **resolvent**
  - C_2 : $(L_1 \mid L_2 \mid ... \mid L_n)$
- Performing all possible applications of unit resolution is called **Boolean Constraint Propagation (BCP)**
- (I'm glossing over one detail: normal forms / CNF)

The DPLL Algorithm

```
bool DPLL (Formula F):
  F' = BCP(F)
  if F' = 1:
    return SAT
  else if F' = 0:
    return UNSAT
  else:
    p = ChooseVariable(F')
    if DPLL(F'[p := 1]):
      return SAT
    else:
      return DPLL(F'[p := 0])
```

Some Refinements to DPLL

- What is “ChooseVariable”?
  - How do we choose?
    - Random guess
    - Use a heuristic
  - Many different heuristics
  - A good one: Variable State Independent Decaying Sum (VSIDS)
  - Each variable has an “activity” that is increased if the variable is involved in a conflict (unsatisfiable clause)
  - Activity is periodically decayed by multiplying by some constant k, $0 < k < 1$
  - ChooseVariable picks the variable with highest activity

Some Refinements to DPLL

- Conflict-Driven Clause Learning
  - Guessing an assignment can lead to a conflict – unsatisfiable under the guess we made
  - Avoid making the same mistake again!
  - We can “learn” a new clause that must also be satisfied
  - e.g. first guess is “p = 1”, but formula is UNSAT before we guess again; learn the clause “p = 0”
  - This prunes the search space

Solving Sudoku with a SAT Solver

- We need to formalize the puzzle as a Boolean formula
  - A set of constraints, all of which must be satisfied
  - C_1 & C_2 & ... & C_n
- Encode the value in each square as nine variables:
  - The square in row i, column j has value v (for 1 <= v <= 9) iff $x_{i,j:v} = 1$

Solving Sudoku with a SAT Solver

- Every square holds some value
  - $x_{1,1:1}$ | $x_{1,1:2}$ | ... | $x_{1,1:9}$
- Every square holds exactly one value
  - $x_{1,1:1}$ -> $!x_{1,1:2}$ ...
- Every square in the same row is different
  - $x_{1,1:1}$ -> $!x_{1,2:1}$ ...
- Every square in the same column is different
  - $x_{1,1:1}$ -> $!x_{2,1:1}$ ...
- Every square in a 3x3 subgrid is different
  - $x_{1,1:1}$ -> $!x_{3,3:1}$ ...
- Some squares have known values (from the puzzle)
  - $x_{1,1:1}$ (if the puzzle gives us a 1 in row 1, column 1)

Solving Sudoku with a SAT Solver

- We can give this to a SAT solver, and solve Sudoku every time!
- The interpretation we find will give us the actual solution
- It will also tell us if a puzzle can't be solved!

Can We Do “Better”?

- Lots of clauses just to specify the range of legal values for one square (x81)
- Lots of clauses to say “these two squares aren't equal”
- This is correct...but not very elegant
- Converting to propositional logic is clunky
  - and hard to maintain / debug
- Something with more expressive power...

Something Worse

- Find natural numbers a, b, c, d such that
  - $2a > b + c$
  - $2b > c + d$
  - $2c > 3d$
  - $3d > a + c$
- Apply the same strategy again:
  - a has value n iff $a_n = 1$
- Then convert > and + to propositional terms
- What could possibly go wrong?!

Something Worse

- There are infinitely many natural numbers.
- We need an infinite number of variables.
- This is not allowed in Boolean logic.

Something Even Worse

This is a modified benchmark from the Flyspeck Project (formal proof of the Kepler Conjecture):

$$\exists^{[3.0,3.14]}\, x_1 .\ \exists^{[-7.0,5.0]}\, x_2 .\ 2 \times 3.14159265 - 2x_1 \arcsin\left(\cos 0.797 \times \sin\left(\frac{3.14159265}{x_1}\right)\right) \leq -0.591 - 0.0331x_2 + 0.506 + 1.0$$

Notice that this is a (first-order) nonlinear inequality over the real numbers.
In general, the satisfiability/validity of such formulas is **undecidable**.

SMT


SATISFIABILITY MODULO THEORIES
S
M
T
SATISFIABILITY MODULO THEORIES

What is SMT?

- Satisfiability Modulo Theories
  - theory of integers, bit-vectors, arrays, reals...
- Combine a SAT solver with theory solvers
  - similar to a constraint solver, with SAT capabilities
- Motivations
  - Easier to encode problems
  - Easier to exploit logic structure / optimize
- DPLL(T) architecture
  - “Purify” each literal into a single theory
  - Set up shared variables to link theories
  - Check satisfiability in each theory
  - Exchange equalities over shared variables

Sudoku in SMT

- Use operations from theory of integers
  - Equality
  - Less than
  - Greater than
- Generate code in SMT-LIBv2 format
  - portable representation for SMT instances
- Try it yourself!

  https://github.com/mtrberzi/sudoku2smt

Sudoku in SMT

- Variables: x11, x12, x19, x21, ...
  - `(declare-const x11 Int)`
- Value specified by puzzle?
  - `(assert (= x11 4))`
- Value not specified?
  - `(assert (>= x11 1)) (assert (<= x11 9))`
- For each pair of values u, v in (same row, same column, same 3x3 square):
  - `(assert (not (= u, v)))`

Sudoku in SMT

- Generate SMT2 expressions for a puzzle
  - November 21, 2008 issue of Imprint, 24 givens
- Statistics:
  - 1517 SMT2 expressions (40KB of text)
  - Solver (Z3) finds the solution in 0.44 seconds
- For an “extremely difficult” puzzle (28 givens):
  - HoDoKu heuristic solver takes >10 seconds
  - Z3 solves in 0.45 seconds

The Z3 SMT Solver

- High-performance general purpose solver
- Microsoft Research
  - z3.codeplex.com
- Free for personal/academic use
- Many theories
  - Linear real/integer arithmetic
  - Bitvectors
  - Uninterpreted functions
  - Arrays
  - Quantifiers
- C/C++, .NET, OCaml, Python, Java, F# APIs
- Program verification: Spec#, HAVOC, VCC, Boogie
- Part of the Static Driver Verifier in the Windows 7 DDK

STP: Bitvector/Array Solver

- Project founder: Dr. Vijay Ganesh (UWaterloo!)
  - V. Ganesh and D.L. Dill. A decision procedure for bit-vectors and arrays. Computer Aided Verification 2007: 519-531.
  - stp.github.io/stp
- Solves constraints generated by program analysis tools
  - Naturally applicable to bug finding and verification
- Very widely used
  - KLEE symbolic fuzzer
  - Stanford, Berkeley, MIT, NVIDIA, “a major microprocessor company”, Certain Government Agencies
  - Has found bugs in mplayer, evince, coreutils, crypto hash implementations
- Extremely high performance
  - 2nd place in the bitvector category at SMT-Comp 2014
  - 1st place in the bitvector category at SMT-Comp 2010 and 2006
  - On a 412 MB input formula with 2.12 million 32-bit variables, array write terms that are tens of thousands of levels deep, array reads with non-constant indices, **STP solves in 2 minutes**
- MIT License

dReal: Real Formula Solver

- An SMT solver for first-order (quantified) nonlinear formulas over real numbers
  - dreal.github.io
- dReal uses a trick called "delta-completeness":
  - Satisfied to within a small error perturbation
  - Can use numerical techniques and symbolic approaches
- GPL license

$$\exists^{[3.0,3.14]} x_1 . \exists^{[-7.0,5.0]} x_2 . 2 \times 3.14159265 - 2x_1 \arcsin\left(\cos 0.797 \times \sin\left(\frac{3.14159265}{x_1}\right)\right) \leq -0.591 - 0.0331x_2 + 0.506 + 1.0$$

- Remember this formula from earlier?
  - It's unsatisfiable; dReal proves this in less than one second

Finding Hash Collisions

- Quick review
  - map data of arbitrary size to a fixed-size value called the “hash”
  - changing the original data will change the hash
  - used in crypto for data validation, etc.
- This example from “Reverse Engineering for Beginners” by Dennis Yurichev
  - beginners.re
  - we skip some of the decompilation and reversing

Finding Hash Collisions

```
#define C1 0x5d7e0d1f2e0f1f84
#define C2 0x388d76aee8cb1500
#define C3 0xd2e9ee7e83c4285b
uint64_t hash(uint64_t v) {
  v = v * C1;
  v = _lrotr(v, v&0xf); // rotate right
  v = v ^ C2;
  v = _lrotl(v, v&0xf); // rotate left
  v = v + C3;
  v = _lrotl(v, v%60); // rotate left
  return v;
}
```

- We want to find two different values for v so that hash(v1) = hash(v2)
- We're not cryptanalysts, so we won't try to break the hash that way
- Brute-force is out of the question, since values are 64 bits
- Can represent this with theory of bitvectors

Finding Hash Collisions

```
from z3 import *
C1=0x5D7E0D1F2E0F1F84
C2=0x388D76AEE8CB1500
C3=0xD2E9EE7E83C4285B
inp, i1, i2, i3, i4, i5, i6, outp = BitVecs('inp i1 i2 i3 i4 i5 i6 
outp', 64)
s = Solver()
s.add(i1==inp*C1)
s.add(i2==RotateRight (i1, i1 & 0xF))
s.add(i3==i2 ^ C2)
s.add(i4==RotateLeft(i3, i3 & 0xF))
s.add(i5==i4 + C3)
s.add(outp==RotateLeft (i5, URem(i5, 60)))
s.add(outp==10816636949158156260)
print s.check()
m=s.model()
print m
print (" inp=0x%X" % m[inp].as_long())
print ("outp=0x%X" % m[outp].as_long())
```

- Implement the algorithm directly in Z3 Python API
- Determine satisfiability and find a satisfying model (given output, find input)
- Z3 finds a model in 0.326 seconds

Finding Hash Collisions

```
from z3 import *
C1=0x5D7E0D1F2E0F1F84
C2=0x388D76AEE8CB1500
C3=0xD2E9EE7E83C4285B
inp, i1, i2, i3, i4, i5, i6, outp = BitVecs('inp i1 i2 i3 i4 i5 i6
outp', 64)
s = Solver()
s.add(i1==inp*C1)
s.add(i2==RotateRight (i1, i1 & 0xF))
s.add(i3==i2 ^ C2)
s.add(i4==RotateLeft(i3, i3 & 0xF))
s.add(i5==i4 + C3)
s.add(outp==RotateLeft (i5, URem(i5, 60)))
s.add(outp==10816636949158156260)
s.add(inp!=0x12EE577B63E80B73)
print s.check()
m=s.model()
print m
print (" inp=0x%X" % m[inp].as_long())
print ("outp=0x%X" % m[outp].as_long())
```

- Use the input we found last time as a constraint (inp != ...) to find a different input
- Now this will find a collision
- Z3 finds one in 0.328 seconds

Finding Hash Collisions

```
from z3 import *
C1=0x5D7E0D1F2E0F1F84
C2=0x388D76AEE8CB1500
C3=0xD2E9EE7E83C4285B
inp, i1, i2, i3, i4, i5, i6, outp = BitVecs('inp i1 i2 i3 i4 i5 i6 outp', 64)
s = Solver()
s.add(i1==inp*C1)
s.add(i2==RotateRight (i1, i1 & 0xF))
s.add(i3==i2 ^ C2)
s.add(i4==RotateLeft(i3, i3 & 0xF))
s.add(i5==i4 + C3)
s.add(outp==RotateLeft (i5, URem(i5, 60)))
s.add(outp==10816636949158156260)
result=[]
while True:
    if s.check() == sat:
        m = s.model()
        print m[inp]
        result.append(m)
        block = []
        for d in m:
            c=d()
            block.append(c != m[d])
        s.add(Or(block))
    else:
            print "results total=",len(result)
            break
```

- Iteratively find *all* inputs that map to this output
- Essentially, after finding a satisfying input, disallow it and run again, until UNSAT
- Z3 finds all 16 inputs that map to this output in 0.689 seconds

Finding Hash Collisions

- This was an example of “symbolic execution”
- Many tools to do this
  - KLEE: symbolic virtual machine
  - EXE: automatically generates inputs of death
  - CATCHCONV: finds type mismatch bugs
- All of these tools use SAT/SMT

Timsort: There Is A Bug

- Broken implementation of sorting algorithm
  - Affects Java, Python, Android
  - Invariant is not maintained during sort
- Formally proven incorrect using KeY (object-oriented verification)
  - http://envisage-project.eu/proving-android-java-and-python-sorting-algorithm-is-broken-and-how-to-fix-it/
  - Uses a SAT or SMT solver as its backend...but there's no documentation

Microfluidic Circuit Design

- Chemical synthesis/analysis with small volumes of fluid
- Currently designed by hand, trial and error
- Design automation
  - Specify the behaviour
  - Generate constraints
  - Use SMT solver
- Relies heavily on nonlinear theories of reals
- For more info, come to our FYDP talk...

Combined Hardware/Software Embedded Systems Analysis

- Model hardware and software together
- Bit-level behaviour of CPU, memory
  - Theory of bitvectors and bitvector operations
  - Theory of arrays
- “Simulate” hardware as it executes a program
- Find an input sequence with desired behaviour
- Exploit hardware bugs or deep internal state

Combined Hardware/Software Embedded Systems Analysis

- Definitions of some terms:
  - **speedrun:** a playthrough of a video game with the intent of completing it as fast as possible
  - **tool-assisted speedrun:** a speedrun that is produced by means of emulation such as slow-motion, frame advance, and re-recording
- The **TAS problem**: given a video game and an integer *n*, find a sequence of inputs that completes the game in at most *n* frames (or find that this is not possible)
- This reduces to the **bounded halting problem**
  - NP-complete

TAS is SAT Spelled Backwards

- Ultimate goal: solve the TAS problem for some game(s)
  - construct representation of game hardware and software for SMT solver
  - look for a satisfying input over *n* input frames
- Side goals: improve state of the art for this problem
  - find new optimizations for this class of instances
  - develop tools that are usable for more general embedded systems
  - produce useful results or difficult benchmarks
    - motivates SAT and SMT solver improvements
- Could have huge implications for:
  - formal methods / program checking
  - symbolic execution / automated bug detection
  - hardware verification

Conclusion

- SAT is a fundamental problem in CS
  - a “classic” hard problem
- SMT is the next generation of SAT
- Many solvers and tools
- Widely applicable to many problem domains