Satisfiability Modulo Theories and Z3

Nikolaj Bjørner
Microsoft Research
ReRISE Winter School, Linz, Austria
February 3, 2014
SMT : Basic Architecture

Sat

+ Theory Solvers

= SMT

- Equality + UF
- Arithmetic
- Bit-vectors
...

Theory[Alcohol]:
Sober $\otimes$ Drunk

Theory[Moodswings]:
Somber $\otimes$ Happy

Nikolaj is Sober $\lor$
Nikolaj is Somber $\lor$
(Nikolaj is Drunk $\land$
Nikolaj is Happy)
Plan

Mon  An invitation to SMT with Z3

Tue  Equalities and Theory Combination

Wed  Theories: Arithmetic, Arrays, Data types

Thu  Quantifiers and Theories

Fri  Programming Z3: Interfacing and Solving
Part 1

I. Satisfiability Modulo Theories in a nutshell

II. SMT solving in a nutshell

III. SMT by example
Takeaways:

• Modern SMT solvers are a often good fit for program analysis tools.
  – Handle domains found in programs directly.

• The selected examples are intended to show instances where sub-tasks are reduced to SMT/Z3.
Wasn’t that easy?

Doctor Rustan’s tool to the rescue

Get to know how debugging your code gets the simple, look and feel of spell checking in Word.

Doctor Rustan’s tool to the rescue

Get to know how debugging your code gets the simple, look and feel of spell checking in Word.

Get to know how debugging your code gets the simple, look and feel of spell checking in Word.

Get to know how debugging your code gets the simple, look and feel of spell checking in Word.

Get to know how debugging your code gets the simple, look and feel of spell checking in Word.

Get to know how debugging your code gets the simple, look and feel of spell checking in Word.

Get to know how debugging your code gets the simple, look and feel of spell checking in Word.

Get to know how debugging your code gets the simple, look and feel of spell checking in Word.

Get to know how debugging your code gets the simple, look and feel of spell checking in Word.

Get to know how debugging your code gets the simple, look and feel of spell checking in Word.

Get to know how debugging your code gets the simple, look and feel of spell checking in Word.

Get to know how debugging your code gets the simple, look and feel of spell checking in Word.

Get to know how debugging your code gets the simple, look and feel of spell checking in Word.

Get to know how debugging your code gets the simple, look and feel of spell checking in Word.

Get to know how debugging your code gets the simple, look and feel of spell checking in Word.

Get to know how debugging your code gets the simple, look and feel of spell checking in Word.

Get to know how debugging your code gets the simple, look and feel of spell checking in Word.

Get to know how debugging your code gets the simple, look and feel of spell checking in Word.

Get to know how debugging your code gets the simple, look and feel of spell checking in Word.

Get to know how debugging your code gets the simple, look and feel of spell checking in Word.

Get to know how debugging your code gets the simple, look and feel of spell checking in Word.

Get to know how debugging your code gets the simple, look and feel of spell checking in Word.

Get to know how debugging your code gets the simple, look and feel of spell checking in Word.

Get to know how debugging your code gets the simple, look and feel of spell checking in Word.

Get to know how debugging your code gets the simple, look and feel of spell checking in Word.

Get to know how debugging your code gets the simple, look and feel of spell checking in Word.

Get to know how debugging your code gets the simple, look and feel of spell checking in Word.

Get to know how debugging your code gets the simple, look and feel of spell checking in Word.

Get to know how debugging your code gets the simple, look and feel of spell checking in Word.

Get to know how debugging your code gets the simple, look and feel of spell checking in Word.

Get to know how debugging your code gets the simple, look and feel of spell checking in Word.

Get to know how debugging your code gets the simple, look and feel of spell checking in Word.

Get to know how debugging your code gets the simple, look and feel of spell checking in Word.

Get to know how debugging your code gets the simple, look and feel of spell checking in Word.

Get to know how debugging your code gets the simple, look and feel of spell checking in Word.

Get to know how debugging your code gets the simple, look and feel of spell checking in Word.

Get to know how debugging your code gets the simple, look and feel of spell checking in Word.

Get to know how debugging your code gets the simple, look and feel of spell checking in Word.

Get to know how debugging your code gets the simple, look and feel of spell checking in Word.

Get to know how debugging your code gets the simple, look and feel of spell checking in Word.

Get to know how debugging your code gets the simple, look and feel of spell checking in Word.

Get to know how debugging your code gets the simple, look and feel of spell checking in Word.

Get to know how debugging your code gets the simple, look and feel of spell checking in Word.

Get to know how debugging your code gets the simple, look and feel of spell checking in Word.

Get to know how debugging your code gets the simple, look and feel of spell checking in Word.

Get to know how debugging your code gets the simple, look and feel of spell checking in Word.

Get to know how debugging your code gets the simple, look and feel of spell checking in Word.

Get to know how debugging your code gets the simple, look and feel of spell checking in Word.

Get to know how debugging your code gets the simple, look and feel of spell checking in Word.

Get to know how debugging your code gets the simple, look and feel of spell checking in Word.

Get to know how debugging your code gets the simple, look and feel of spell checking in Word.

Get to know how debugging your code gets the simple, look and feel of spell checking in Word.

Get to know how debugging your code gets the simple, look and feel of spell checking in Word.

Get to know how debugging your code gets the simple, look and feel of spell checking in Word.

Get to know how debugging your code gets the simple, look and feel of spell checking in Word.

Get to know how debugging your code gets the simple, look and feel of spell checking in Word.

Get to know how debugging your code gets the simple, look and feel of spell checking in Word.

Get to know how debugging your code gets the simple, look and feel of spell checking in Word.

Get to know how debugging your code gets the simple, look and feel of spell checking in Word.

Get to know how debugging your code gets the simple, look and feel of spell checking in Word.
Satisfiability Modulo Theories: Introduction & Applications

Leonardo de Moura
Microsoft Research
One Microsoft Way
Redmond, WA 98052
leonardo@microsoft.com

Nikolaj Bjørner
Microsoft Research
One Microsoft Way
Redmond, WA 98052
nbjorner@microsoft.com

Abstract:

SMT solvers have been the focus of increased recent attention thanks to technological advances and industrial applications. Yet, they draw on a combination of some of the most fundamental areas in computer science as well as discoveries from the past century of symbolic logic. They combine the problem of Boolean Satisfiability with domains, such as, those studied in convex optimization and term-manipulating symbolic systems. They involve the decision problem, completeness and incompleteness of logical theories, and finally complexity theory. In this article, we present an overview of the field of Satisfiability Modulo Theories, and some of its applications.

1.1 An SMT Application - Scheduling

Consider the classical job shop scheduling decision problem. In this problem, there are \( n \) jobs, each composed of \( m \) tasks of varying duration that have to be performed consecutively on \( m \) machines. The start of a new task can be delayed as long as needed in order to wait for a machine to become available, but tasks cannot be interrupted once they start.
Some Microsoft Tools based on Z3

- Program Verification
- Auditing
- Type Safety
- Over-Approximation
- Under-Approximation
- Testing
- Analysis
- Synthesis
### Software Engineering Tools

<table>
<thead>
<tr>
<th>Tool</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fe</td>
<td>A verification tool for higher-order programs</td>
</tr>
<tr>
<td>fast</td>
<td>A domain-specific language for writing and analyzing tree manipulating programs</td>
</tr>
<tr>
<td>le3</td>
<td>Efficient Implementation Theorem Prover</td>
</tr>
<tr>
<td>agl</td>
<td>Automatic Graph Layout</td>
</tr>
<tr>
<td>bek</td>
<td>A domain-specific language for writing and analyzing common integer operations</td>
</tr>
<tr>
<td>bex</td>
<td>A domain-specific language for writing and analyzing string operations</td>
</tr>
<tr>
<td>boogie</td>
<td>Intermediate Verification Language</td>
</tr>
<tr>
<td>boogie</td>
<td>Boogie: Intermediate Verification Language</td>
</tr>
<tr>
<td>chalice</td>
<td>A language and program verifier for reasoning about concurrent programs</td>
</tr>
<tr>
<td>code contracts</td>
<td>Code contracts for modular program verification and repair with abstract interpretation</td>
</tr>
<tr>
<td>counterdog</td>
<td>Theorem-prover for Counter-Factual Analysis</td>
</tr>
<tr>
<td>dafny</td>
<td>A language and program verifier for functional correctness</td>
</tr>
<tr>
<td>dkal</td>
<td>Distributed Knowledge Authorization Language</td>
</tr>
<tr>
<td>eism</td>
<td>ESIM: Software Engineering and Measurement Group</td>
</tr>
<tr>
<td>fast</td>
<td>A domain-specific language for writing and analyzing tree manipulating programs</td>
</tr>
<tr>
<td>formula</td>
<td>Formal Model Checking Logic Programming and Analysis</td>
</tr>
<tr>
<td>formula2</td>
<td>Formal Model Checking Logic Programming and Analysis</td>
</tr>
<tr>
<td>try f#</td>
<td>Programming language combining functional and object-oriented paradigms</td>
</tr>
<tr>
<td>rex</td>
<td>Regular Expression Exploration</td>
</tr>
<tr>
<td>seal</td>
<td>S2E: Static Analysis</td>
</tr>
<tr>
<td>slayer</td>
<td>Static Analysis</td>
</tr>
<tr>
<td>spec#</td>
<td>A formal language for ASP contracts</td>
</tr>
<tr>
<td>touch develop</td>
<td>Touch develop for deploying and running applications</td>
</tr>
<tr>
<td>vcc</td>
<td>A verifier for Concurrent C</td>
</tr>
<tr>
<td>visual c++</td>
<td>The Visual C++ Compiler</td>
</tr>
</tbody>
</table>

### Institutions

- Albert-Ludwigs-Universität Freiburg
- ETH Zurich - Chair of Software Engineering
- KU Leuven
- Multicore Programming Group, Imperial College London
- University of Utah and IMDEA Software Institute
SMT IN A NUTSHELL
Is formula $\varphi$ satisfiable modulo theory $T$?

SMT solvers have specialized algorithms for $T$. 

SMT solvers have specialized algorithms for $T$. 

Satisfiability Modulo Theories (SMT)

\[ x + 2 = y \Rightarrow f(\text{select}(\text{store}(a, x, 3), y - 2)) = f(y - x + 1) \]

Array Theory
Arithmetic
Uninterpreted Functions

\[
\begin{align*}
\text{select}(\text{store}(a, i, v), i) &= v \\
i \neq j &\Rightarrow \text{select}(\text{store}(a, i, v), j) = \text{select}(a, j)
\end{align*}
\]
SMT SOLVING IN A NUTSHELL

Job Shop Scheduling
Job Shop Scheduling

Machines

Tasks

Jobs

\[ P = \text{NP?} \]

\[ \zeta(s) = 0 \Rightarrow s = \frac{1}{2} + ir \]
Job Shop Scheduling

Constraints:

**Precedence:** between two tasks of the same job

**Resource:** Machines execute at most one job at a time

\[
[\text{start}_{2,2}..\text{end}_{2,2}] \cap [\text{start}_{4,2}..\text{end}_{4,2}] = \emptyset
\]
Job Shop Scheduling

Constraints:

Precedence:

Resource:

[\text{start}_{2,2}..\text{end}_{2,2}] \cap [\text{start}_{4,2}..\text{end}_{4,2}] = \emptyset

Encoding:

\begin{align*}
t_{2,3} & \quad \text{- start time of job 2 on mach 3} \\
d_{2,3} & \quad \text{- duration of job 2 on mach 3} \\
t_{2,3} + d_{2,3} & \leq t_{2,4}
\end{align*}

Not convex

\begin{align*}
t_{2,2} + d_{2,2} & \leq t_{4,2} \\
\lor \\
t_{4,2} + d_{4,2} & \leq t_{2,2}
\end{align*}
Job Shop Scheduling

<table>
<thead>
<tr>
<th>$d_{i,j}$</th>
<th>Machine 1</th>
<th>Machine 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Job 1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Job 2</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Job 3</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

$max = 8$

**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**

$(t_{1,1} \geq 0) \land (t_{1,2} \geq t_{1,1} + 2) \land (t_{1,2} + 1 \leq 8) \land$

$(t_{2,1} \geq 0) \land (t_{2,2} \geq t_{2,1} + 3) \land (t_{2,2} + 1 \leq 8) \land$

$(t_{3,1} \geq 0) \land (t_{3,2} \geq t_{3,1} + 2) \land (t_{3,2} + 3 \leq 8) \land$

$((t_{1,1} \geq t_{2,1} + 3) \lor (t_{2,1} \geq t_{1,1} + 2)) \land$

$((t_{1,1} \geq t_{3,1} + 2) \lor (t_{3,1} \geq t_{1,1} + 2)) \land$

$((t_{2,1} \geq t_{3,1} + 2) \lor (t_{3,1} \geq t_{2,1} + 3)) \land$

$((t_{1,2} \geq t_{2,2} + 1) \lor (t_{2,2} \geq t_{1,2} + 1)) \land$

$((t_{1,2} \geq t_{3,2} + 3) \lor (t_{3,2} \geq t_{1,2} + 1)) \land$

$((t_{2,2} \geq t_{3,2} + 3) \lor (t_{3,2} \geq t_{2,2} + 1))$
Job Shop Scheduling

\[
(t_{1,1} \geq 0) \land (t_{1,2} \geq t_{1,1} + 2) \land (t_{1,2} + 1 \leq 8) \land \\
(t_{2,1} \geq 0) \land (t_{2,2} \geq t_{2,1} + 3) \land (t_{2,2} + 1 \leq 8) \land \\
(t_{3,1} \geq 0) \land (t_{3,2} \geq t_{3,1} + 2) \land (t_{3,2} + 3 \leq 8) \land \\
((t_{1,1} \geq t_{2,1} + 3) \lor (t_{2,1} \geq t_{1,1} + 2)) \land \\
((t_{1,1} \geq t_{3,1} + 2) \lor (t_{3,1} \geq t_{1,1} + 2)) \land \\
((t_{2,1} \geq t_{3,1} + 2) \lor (t_{3,1} \geq t_{2,1} + 3)) \land \\
((t_{1,2} \geq t_{2,2} + 1) \lor (t_{2,2} \geq t_{1,2} + 1)) \land \\
((t_{1,2} \geq t_{3,2} + 3) \lor (t_{3,2} \geq t_{1,2} + 1)) \land \\
((t_{2,2} \geq t_{3,2} + 3) \lor (t_{3,2} \geq t_{2,2} + 1))
\]

Efficient solvers:
- Floyd-Warshall algorithm
- Ford-Fulkerson algorithm

\[
z - t_{1,1} \leq 0 \\
z - t_{2,1} \leq 0 \\
z - t_{3,1} \leq 0 \\
t_{3,2} - z \leq 5 \\
t_{3,1} - t_{3,2} \leq -2 \\
t_{2,1} - t_{3,1} \leq -3 \\
t_{1,1} - t_{2,1} \leq -2
\]

\[
z - z = 5 - 2 - 3 - 2 = -2 < 0
\]
THEORIES
Theories

Uninterpreted functions
Theories: z3py

Explore the Z3 API using Python

```python
1 t11, t12, t21, t22, t31, t32 = Ints('t11 t12 t21 t22 t31 t32')
2
3 s = Solver()
4
5 s.add(And([t11 >= 0, t12 >= t11 + 2, t12 + 1 <= 8]))
6 s.add(And([t21 >= 0, t22 >= t21 + 3, t22 + 1 <= 8]))
7 s.add(And([t31 >= 0, t32 >= t31 + 2, t32 + 3 <= 8]))
8
9 s.add(Or(t11 >= t21 + 3, t21 >= t11 + 2))
10 s.add(Or(t11 >= t31 + 2, t31 >= t11 + 2))
11 s.add(Or(t21 >= t31 + 2, t31 >= t21 + 3))
12 s.add(Or(t21 >= t22 + 1, t22 >= t12 + 1))
13 s.add(Or(t12 >= t32 + 3, t32 >= t12 + 1))
14 s.add(Or(t22 >= t32 + 3, t32 >= t22 + 1))
15 |
16 print ">>", s.check()
17 print ">>", s.model()
18
19
```

Uninterpreted function

Arithmetic (linear)
Theories

Uninterpreted functions

Arithmetic (linear)

Bit-vectors

```python
x = BitVec('x', 32)
powers = [ 2**i for i in range(32) ]
fast = And(x != 0, x & (x - 1) == 0)
slow = Or([ x == p for p in powers ])

prove(fast == slow)
print "buggy version..."

fast = x & (x - 1) == 0
prove(fast == slow)
```

proven

buggy version...

counterexample

[x = 0]
Theories

- Uninterpreted functions
- Arithmetic (linear)
- Bit-vectors
- Algebraic data-types
Theories

Uninterpreted functions
Arithmetic (linear)
Bit-vectors
Algebraic data-types
Arrays
Theories

Uninterpreted functions
Arithmetic (linear)
Bit-vectors
Algebraic data-types
Arrays

Polynomial Arithmetic

# z3py

Explore the Z3 API using Python

```python
1 x, y, z = Reals('x y z')
2
3 solve(x**2 + y**2 < 1, x*y > 1,
        show=True)
4
5 solve(x**2 + y**2 < 1, x*y > 0.4,
        show=True)
6
7 solve(x**2 + y**2 < 1, x*y > 0.4, x < 0,
        show=True)
8
9 solve(x**5 - x - y == 0, Or(y == 1, y == -1),
        show=True)
10
11
12 solve(x**5 - x - y == 0, Or(y == 1, y == -1),
        show=True)
13
14
```
QUANTIFIERS
Equality-Matching

\[ p(\forall \ldots) \]
\[ \land \quad a = g(b, b) \]
\[ \land \quad b = c \]
\[ \land \quad f(a) \neq c \]
\[ \land \quad p(\forall x \ldots) \rightarrow f(g(c, b)) = b \]

\( g(c, x) \) matches \( g(b, b) \)
with substitution \([x \mapsto b]\)
modulo \( b = c \)

[de Moura, B. CADE 2007]
Quantifier Elimination

Presburger Arithmetic,
Algebraic Data-types,
Quadratic polynomials

SMT integration to prune branches

[B. IJCAR 2010]
MBQI: Model based Quantifier Instantiation

(set-option :mbqi true)
(declare-fun f (Int Int) Int)
(declare-const a Int)
(declare-const b Int)

(assert (forall ((x Int)) (>= (f x x) (+ x a))))

(assert (< (f a b) a))
(assert (> a 0))
(check-sat)
(get-model)

(echo "evaluating (f (+ a 10) 20)"
(eval (f (+ a 10) 20))

[de Moura, Ge. CAV 2008]
[Bonachnia, Lynch, de Moura CADE 2009]
[de Moura, B. IJCAR 2010]
Horn Clauses

\[ \begin{align*}
mc(x) &= x - 10 & \text{if } x > 100 \\
mc(x) &= mc(mc(x + 11)) & \text{if } x \leq 100
\end{align*} \]

assert \((mc(x) \geq 91)\)

\[ \begin{align*}
\forall X. & \ X > 100 \rightarrow mc(X, X - 10) \\
\forall X, Y, R. & \ X \leq 100 \land mc(X + 11, Y) \land mc(Y, R) \rightarrow mc(X, R) \\
\forall X, R. & \ mc(X, R) \land X \leq 101 \rightarrow R = 91
\end{align*} \]

*Solver finds solution for mc*
MODELS, PROOFS, CORES & SIMPLIFICATION
Models

Logical Formula

Click on a tool to load a sample then ask!

(define-sorts ((A (Array Int Int))))
(declare-funs ((x Int) (y Int) (z Int)))
(declare-funs ((a1 A) (a2 A) (a3 A)))
(assert (= (select a1 x) x))
(assert (= (store a1 x y) a1))
(check-sat)
(get-info model)

Is this SMT formula satisfiable?
Click 'ask Z3'! Read more or watch the video.

sat
("model" "
(define x 0)
(define a1 as-array[k!0])
(define y 0)
(define (k!0 (x1 Int))
(if (= x1 0) 0 1))")
Simplification

R1SE4fun

Simplify Logical Formula

(declare-fun x () Real)
(declare-fun y () Real)
(simplify (>= x (+ x y)))

ask z3

Is this SMT formula satisfiable? Click 'ask z3'! Read more or watch the video.

(<= y 0.0)

Microsoft Research R1SE
Cores

Logical Formula

```lisp
(declare-preds ((p) (q) (r) (s)))
(set-option enable-cores)
(assert (or p q))
(assert (implies r s))
(assert (implies s (iff q r)))
(assert (or r p))
(assert (or r s))
(assert (not (and r q)))
(assert (not (and s p)))
(check-sat)
(get-unsat-core)
```

Unsat. Core

```lisp
unsat
((or p q)
 (=> r s)
 (or r p)
 (or r s)
 (not (and r q))
 (not (and s p)))
```
TACTICS, SOLVERS
Tactics

(declare-const x (_ BitVec 16))
(declare-const y (_ BitVec 16))

(assert (= (bvor x y) (_ bv13 16)))
(assert (bvslt x y))

(check-sat-using (then simplify solve-eqs bit-blast sat))
(get-model)

Composition of tactics:
• (then t s)
• (par-then t s) applies t to the input goal and S to every subgoal produced by t in parallel.
• (or-else t s)
• (par-or t s) applies t and S in parallel until one of them succeed.
• (repeat t)
• (repeat t n)
• (try-for t ms)
• (using-params t params) Apply the given tactic using the given parameters.
Solvers

- Tactics take goals and reduce to sub-goals
- Solvers take tactics and serve as logical contexts.
  - push
  - add
  - check
  - model, core, proof
  - pop

```python
bv_solver = Then(With('simplify', mul2concat=True),
                 'solve-eqs',
                 'bit-blast',
                 'aig',
                 'sat').solver()

x, y = BitVecs('x y', 16)
bv_solver.add(x*32 + y == 13, x & y < 10, y > -100)
print bv_solver.check()
m = bv_solver.model()
print m
print x*32 + y, '==', m.evaluate(x*32 + y)
print x & y, '==', m.evaluate(x & y)
```
APIS

- C
- C++
- python
- OCaml
- Java
- .NET
SMT : Basic Architecture

SAT + Theory Solvers = SMT

- Equality + UF
- Arithmetic
- Bit-vectors
- ...

Case Analysis
SAT + Theory solvers

Basic Idea

\[ x \geq 0, \ y = x + 1, \ (y > 2 \lor y < 1) \]

Abstract (aka “naming” atoms)

\[ p_1, \ p_2, \ (p_3 \lor p_4) \]

\[ p_1 \equiv (x \geq 0), \ p_2 \equiv (y = x + 1), \ p_3 \equiv (y > 2), \ p_4 \equiv (y < 1) \]
SAT + Theory solvers

Basic Idea

\[ x \geq 0, \ y = x + 1, \ (y > 2 \lor y < 1) \]

Abstract (aka “naming” atoms)

\[ p_1, \ p_2, \ (p_3 \lor p_4) \]

\[ p_1 \equiv (x \geq 0), \ p_2 \equiv (y = x + 1), \]
\[ p_3 \equiv (y > 2), \ p_4 \equiv (y < 1) \]
SAT + Theory solvers

**Basic Idea**

\[ x \geq 0, \ y = x + 1, \ (y > 2 \lor y < 1) \]

Abstract (aka “naming” atoms)

\[ p_1, \ p_2, \ (p_3 \lor p_4) \]

\[ p_1 \equiv (x \geq 0), \ p_2 \equiv (y = x + 1), \]
\[ p_3 \equiv (y > 2), \ p_4 \equiv (y < 1) \]

**SAT Solver**

Assignment

\[ p_1, \ p_2, \ \neg p_3, \ p_4 \]
SAT + Theory solvers

Basic Idea

\[ x \geq 0, y = x + 1, (y > 2 \lor y < 1) \]

Abstract (aka “naming” atoms)

\[ p_1, p_2, (p_3 \lor p_4) \]

\[ p_1 \equiv (x \geq 0), p_2 \equiv (y = x + 1), p_3 \equiv (y > 2), p_4 \equiv (y < 1) \]

SAT Solver

Assignment

\[ p_1, p_2, \neg p_3, p_4 \]

\[ x \geq 0, y = x + 1, \neg(y > 2), y < 1 \]
SAT + Theory solvers

Basic Idea

\[ x \geq 0, \ y = x + 1, \ (y > 2 \lor y < 1) \]

Abstract (aka “naming” atoms)

\[ p_1, \ p_2, (p_3 \lor p_4) \]
\[ p_1 \equiv (x \geq 0), \ p_2 \equiv (y = x + 1), \]
\[ p_3 \equiv (y > 2), \ p_4 \equiv (y < 1) \]

SAT Solver

Assignment

\[ p_1, \ p_2, \neg p_3, \ p_4 \]
\[ x \geq 0, \ y = x + 1, \neg(y > 2), \ y < 1 \]

Theory Solver

Unsatisfiable

\[ x \geq 0, \ y = x + 1, \ y < 1 \]
SAT + Theory solvers

Basic Idea

\[ x \geq 0, \ y = x + 1, \ (y > 2 \lor y < 1) \]

Abstract (aka “naming” atoms)

\[ p_1, \ p_2, \ (p_3 \lor p_4) \]
\[ p_1 \equiv (x \geq 0), \ p_2 \equiv (y = x + 1), \]
\[ p_3 \equiv (y > 2), \ p_4 \equiv (y < 1) \]

SAT Solver

Assignment

\[ p_1, \ p_2, \ \neg p_3, \ p_4 \]
\[ x \geq 0, \ y = x + 1, \]
\[ \neg (y > 2), \ y < 1 \]

New Lemma

\[ \neg p_1 \lor \neg p_2 \lor \neg p_4 \]

Unsatisfiable

\[ x \geq 0, \ y = x + 1, \ y < 1 \]

Theory Solver
SAT + Theory solvers

New Lemma
\neg p_1 \lor \neg p_2 \lor \neg p_4

Unsatisfiable
\ x \geq 0, y = x + 1, y < 1

AKA Theory conflict

Theory Solver
SAT/SMT SOLVING USING DPLL(T)/CDCL
Mile High: Modern SAT/SMT search
## Core Engine in Z3: Modern DPLL/CDCL

<table>
<thead>
<tr>
<th>Step</th>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Initialize</strong></td>
<td>$\epsilon \models F$</td>
<td>$F$ is a set of clauses</td>
</tr>
<tr>
<td><strong>Decide</strong></td>
<td>$M \models F \Rightarrow M, \ell \models F \quad \ell$ is unassigned</td>
<td>$\ell$ is unassigned</td>
</tr>
<tr>
<td><strong>Propagate</strong></td>
<td>$M \models F, C \lor \ell \Rightarrow M, \ell^{C \lor \ell} \models F, C \lor \ell \quad C$ is false under $M$</td>
<td>$C$ is false under $M$</td>
</tr>
<tr>
<td><strong>Sat</strong></td>
<td>$M \models F \Rightarrow M \quad F$ true under $M$</td>
<td></td>
</tr>
<tr>
<td><strong>Conflict</strong></td>
<td>$M \models F, C \Rightarrow M \models F, C \models C \quad C$ is false under $M$</td>
<td>$C$ is false under $M$</td>
</tr>
<tr>
<td><strong>Learn</strong></td>
<td>$M \models F \models C \Rightarrow M \models F, C \models C \quad C$ is false under $M$</td>
<td>$C$ is false under $M$</td>
</tr>
<tr>
<td><strong>Unsat</strong></td>
<td>$M \models F \models \emptyset \Rightarrow \text{Unsat}$</td>
<td></td>
</tr>
<tr>
<td><strong>Backjump</strong></td>
<td>$MM' \models F \models C \lor \ell \Rightarrow M\ell^{C \lor \ell} \models F \quad \bar{C} \subseteq M, \neg \ell \in M'$</td>
<td>$\bar{C} \subseteq M, \neg \ell \in M'$</td>
</tr>
<tr>
<td><strong>Resolve</strong></td>
<td>$M \models F \models C' \lor \neg \ell \Rightarrow M \models F \models C' \lor C \quad \ell^{C \lor \ell} \in M$</td>
<td>$\ell^{C \lor \ell} \in M$</td>
</tr>
<tr>
<td><strong>Forget</strong></td>
<td>$M \models F, C \Rightarrow M \models F \quad C$ is a learned clause</td>
<td></td>
</tr>
<tr>
<td><strong>Restart</strong></td>
<td>$M \models F \Rightarrow \epsilon \models F \quad$ [Nieuwenhuis, Oliveras, Tinelli J.ACM 06] customized</td>
<td></td>
</tr>
</tbody>
</table>
DPLL($\mathcal{T}$) solver interaction

**T- Propagate**
\[ M \mid F, C \lor \ell \implies M, \ell^{C \lor \ell} \mid F, C \lor \ell \quad \text{C is false under } T + M \]

**T- Conflict**
\[ M \mid F \implies M \mid F, \neg M' \quad \text{M' \subseteq M and M' is false under } T \]

**T- Propagate**
\[ a > b, b > c \quad \mid F, a \leq c \lor b \leq d \implies \]
\[ a > b, b > c, b \leq d^{a \leq c \lor b \leq d} \mid F, a \leq c \lor b \leq d \]

**T- Conflict**
\[ M \mid F \implies M \mid F, a \leq b \lor b \leq c \lor c < a \]
\[ \text{where } a > b, b > c, a \leq c \subseteq M \]
Summary

Z3 supports several theories
  – Using a default combination
  – Providing custom tactics for special combinations

Z3 is more than sat/unsat
  – Models, proofs, unsat cores,
  – simplification, quantifier elimination are tactics

Prototype with python/smt-lib2
  – Implement using smt-lib2/programmatic API