SMT Solver Z3

Testing, Quality Assurance, and Maintenance
Winter 2020

Prof. Arie Gurfinkel
Satisfiability Modulo Theory (SMT)

Satisfiability is the problem of determining whether a formula $F$ has a model

- if $F$ is *propositional*, a model is a truth assignment to Boolean variables
- if $F$ is *first-order formula*, a model assigns values to variables and interpretation to all the function and predicate symbols

**SAT Solvers**

- check satisfiability of propositional formulas

**SMT Solvers**

- check satisfiability of formulas in a *decidable* first-order theory (e.g., linear arithmetic, uninterpreted functions, array theory, bit-vectors)
SAT solvers are often used as a core component in a variety of applications, including software verification, hardware verification, and planning. They are often used to solve constraint satisfaction problems, which involve finding a solution that satisfies a set of constraints.

SMT solvers are used in a wide range of applications, including software and hardware verification, planning, and theorem proving. They are particularly useful when dealing with problems that involve both logical and numeric constraints. In this paper, we explore the capabilities of SMT solvers and their applications in detail.

1. Introduction to SMT

Satisfiability Modulo Theories (SMT) is a powerful technique for solving constraint satisfaction problems. It extends the capabilities of SAT solvers by allowing them to reason about additional theories, such as arithmetic, array theory, and bit-vector theory. SMT solvers are used in a wide range of applications, including software and hardware verification, planning, and theorem proving.

1.1 An SMT Application - Scheduling

SMT solvers are often used in scheduling problems, where a set of tasks need to be assigned to a set of resources. SMT solvers can be used to find a feasible schedule that satisfies all the constraints, such as the availability of resources and the duration of tasks.

1.2 Applications in Planning

SMT solvers can also be used in planning problems, where the goal is to find a sequence of actions that leads to a desired state. SMT solvers can be used to find a plan that satisfies all the constraints, such as the availability of resources and the feasibility of actions.

1.3 Applications in Theorem Proving

SMT solvers are also used in theorem proving, where the goal is to prove the validity of a logical formula. SMT solvers can be used to find a proof of a theorem or to find a counterexample, if the theorem is not valid.

2. Related Work

SMT solvers have been developed by a number of research groups, including Microsoft Research, IBM, and Stanford University. Many SMT solvers are available as open-source software, allowing researchers to experiment with different theories and algorithms.

3. Conclusion

SMT solvers are a powerful tool for solving constraint satisfaction problems. They have been used in a wide range of applications, including software and hardware verification, planning, and theorem proving. As the capabilities of SMT solvers continue to grow, we can expect to see even more applications in the future.
Example

\[ b + 2 = c \land f(\text{read}(\text{write}(a, b, 3), c - 2)) \neq f(c - b + 1) \]
Example

\[ b + 2 = c \land f(\text{read}(\text{write}(a, b, 3), c - 2)) \neq f(c - b + 1) \]

Arithmetic
Example

\[ b + 2 = c \land f(\text{read}(\text{write}(a, b, 3), c - 2)) \neq f(c - b + 1) \]

Array theory
Example

\[ b + 2 = c \land f(\text{read}(\text{write}(a, b, 3), c - 2)) \neq f(c - b + 1) \]

Uninterpreted function
Example

\[ b + 2 = c \land f(\text{read}(\text{write}(a, b, 3), c - 2)) \neq f(c - b + 1) \]
Example

\[ b + 2 = c \land f(\text{read}(\text{write}(a, b, 3), c - 2)) \neq f(c - b + 1) \]

By arithmetic, this is equivalent to

\[ b + 2 = c \land f(\text{read}(\text{write}(a, b, 3), b)) \neq f(3) \]
Example

\[ b + 2 = c \land f(\text{read}(\text{write}(a, b, 3), c - 2)) \neq f(c - b + 1) \]

By arithmetic, this is equivalent to

\[ b + 2 = c \land f(\text{read}(\text{write}(a, b, 3), b)) \neq f(3) \]

then, by the array theory axiom: \text{read}(\text{write}(v, i, x), i) = x

\[ b + 2 = c \land f(3) \neq f(3) \]
Example

\[ b + 2 = c \land f(\text{read(}\text{write}(a, b, 3), c - 2)) \neq f(c - b + 1) \]

By arithmetic, this is equivalent to

\[ b + 2 = c \land f(\text{read(}\text{write}(a, b, 3), b)) \neq f(3) \]

then, by the array theory axiom: \text{read(}\text{write}(v, i, x), i) = x

\[ b + 2 = c \land f(3) \neq f(3) \]

then, the formula is unsatisfiable
Example 2

\[ x \geq 0 \land f(x) \geq 0 \land y \geq 0 \land f(y) \geq 0 \land x \neq y \]
Example 2

\[ x \geq 0 \land f(x) \geq 0 \land y \geq 0 \land f(y) \geq 0 \land x \neq y \]

This formula is **satisfiable**
Example 2

\[ x \geq 0 \land f(x) \geq 0 \land y \geq 0 \land f(y) \geq 0 \land x \neq y \]

This formula is **satisfiable**:

Example model:

\[ x \rightarrow 1 \]
\[ y \rightarrow 2 \]
\[ f(1) \rightarrow 0 \]
\[ f(2) \rightarrow 1 \]
\[ f(\ldots) \rightarrow 0 \]
SMT-LIB: http://smt-lib.org

International initiative for facilitating research and development in SMT
Provides rigorous definition of syntax and semantics for theories

SMT-LIB syntax

• based on s-expressions (LISP-like)

• common syntax for interpreted functions of different theories
  – e.g. (and (= x y) (<= (* 2 x) z))

• commands to interact with the solver
  – (declare-fun ...) declares a constant/function symbol
  – (assert p) conjoins formula p to the current context
  – (check-sat) checks satisfiability of the current context
  – (get-model) prints current model (if the context is satisfiable)

• see examples at http://rise4fun.com/z3
SMT-LIB Syntax

(declare-fun x () Int)
(declare-fun y () Int)
(declare-fun z () Int)
(assert (>= (* 2 x) (+ y z)))
(declare-fun f (Int) Int)
(declare-fun g (Int Int) Int)
(assert (< (f x) (g x x)))
(assert (> (f y) (g x x)))
(check-sat)
(get-model)
Is this formula satisfiable?

1 ; This example illustrates basic arithmetic and
2 ; uninterpreted functions
3 
4 (declare-fun x () Int)
5 (declare-fun y () Int)
6 (declare-fun z () Int)
7 (assert (>= (* 2 x) (+ y z)))
8 (declare-fun f (Int) Int)
9 (declare-fun g (Int Int) Int)
10 (assert (< (f x) (g x x)))
11 (assert (> (f y) (g x x)))
12 (check-sat)
13 (get-model)
14 (push)
15 (assert (= x y))
16 (check-sat)
17 (pop)
18 (exit)
19

http://rise4fun.com/z3
Example

\[ b + 2 = c \land f(\text{read}(\text{write}(a, b, 3), c - 2)) \neq f(c - b + 1) \]
Is this formula satisfiable?

1  ;; Is this formula satisfiable?
2  (declare-fun b () Int)
3  (declare-fun c () Int)
4  (declare-fun a () (Array Int Int))
5  (declare-fun f (Int) Int)
6  (assert (= (+ b 2) c))
7  (assert (not (= (f (select (store a b 3) (- c 2))) (f (+ (- c b) 1)))))
8  (check-sat)
import z3

def main():
    b, c = z3.Ints ('b c')
    a = z3.Array ('a', z3.IntSort(), z3.IntSort())
    solver = z3.Solver ()
    solver.add (c == b + z3.IntVal(2))
    lhs = f (z3.Store (a, b, 3)[c-2])
    rhs = f(c-b+1)
    solver.add (lhs <> rhs)
    res = solver.check ()
    if res == z3.sat:
        print 'sat'
    elif res == z3.unsat:
        print 'unsat'
    else:
        print 'unknown'
if __name__ == '__main__':
    main()
import z3

def main():
    b, c = z3.Ints('b c')
    a = z3.Array('a', z3.IntSort(), z3.IntSort())
    solver = z3.Solver()
    solver.add(c == b + z3.IntVal(2))
    lhs = f(z3.Store(a, b, 3)[c-2])
    rhs = f(c-b+1)
    solver.add(lhs <> rhs)
    res = solver.check()
    if res == z3.sat:
        print('sat')
    elif res == z3.unsat:
        print('unsat')
    else:
        print('unknown')

if __name__ == '__main__':
    main()
Useful Z3Py Functions

All these functions are under python package `z3`

Create constants and values

- `Int(name)` – an integer constant with a given name
- `FreshInt(name)` – unique constant starting with name
- `IntVal(v), BoolVal(v)` – integer and boolean values

Arithmetic functions and predicates

- `+, -, /, <, <=, >, >=, ==, etc.`
- `Distinct(a, b, …)` – the arguments are distinct (expands to many disequalities)

Propositional operators

- `And`, `Or`, `Not`

Methods of the `z3.Solver` class

- `add(fml)` – add formula `fml` to the solver
- `check()` – returns `z3.sat`, `z3.unsat`, or `z3.unknown` (on failure to solve)
- `model()` – model if the result is sat

Methods of `z3.Model` class

- `eval(fml)` – returns the value of `fml` in the model
Job Shop Scheduling

P = 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

\[
[start_{2,2}, end_{2,2}] \cap [start_{4,2}, end_{4,2}] = \emptyset
\]
Job Shop Scheduling

Constraints:

**Precedence:**

\[ t_{2,3} \] - start time of job 2 on mach 3

\[ d_{2,3} \] - duration of job 2 on mach 3

\[ t_{2,3} + d_{2,3} \leq t_{2,4} \]

**Resource:**

\[ [start_{2,2} \ldots end_{2,2}] \cap [start_{4,2} \ldots end_{4,2}] = \emptyset \]

Encoding:

\[ t_{2,3} \] - start time of job 2 on mach 3

\[ d_{2,3} \] - duration of job 2 on mach 3

\[ t_{2,3} + d_{2,3} \leq t_{2,4} \]

Not convex

\[ t_{2,2} + d_{2,2} \leq t_{4,2} \]

\[ t_{4,2} + d_{4,2} \leq t_{2,2} \]
Job Shop Scheduling

<table>
<thead>
<tr>
<th></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

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

Spend exactly 100 dollars and buy exactly 100 animals.

- Dogs cost 15 dollars,
- cats cost 1 dollar,
- and mice cost 25 cents each.

You have to buy at least one of each.

How many of each should you buy?
Bit Tricks

Let $x, y$ be a 32 bit machine integers (a bit-vector)

Show that $x \neq 0 \land \neg (x \land (x-1))$ is true iff $x$ is a power of 2

Show that $x$ and $y$ have different signs iff $x^y < 0$
Eight Queens Problem

Place 8 queens on an 8x8 chess board so that no two queen attacks one another
Incremental Interface

Z3 provides two interfaces for incremental solving that allow for adding and removing constraints

- push/pop, and assumptions

Constraints can be added at any time. This is not called incremental 😊

Push/Pop Interface

- Store current solver state by a call to push
  - s.push() in Python, and (push) in SMT-LIB
- Restore previous state by a call to pop
  - s.pop() in Python and (pop) in SMT-LIB
Incremental Interface: Assumptions

Requires two steps, but much more flexible than push/pop

1. tag constraints by fresh Boolean constants
   - e.g., use (assert (=> p phi)) instead of (assert phi)

2. during check-sat, enable constraints by forcing tags to be true
   - e.g., use (check-sat p)

For example,

(assert (=> a0 c0))
(assert (=> a1 c1))
(assert (=> a2 c2))
(check-sat a0) ; check whether c0 is sat
(check-sat a0 a2) ; check whether c0 and c2 are sat
(check-set a1 a2) ; check whether c1 and c3 are sat
Assumptions in Python Interface

Methods of `z3.Solver` class

- `check(self, *assumptions)` — check with assumptions
- `unsat_core(self)` — if the last call to `check` was `unsat`, returns the subset of assumptions that were actually used to show `unsat`