A Short Overview of Satisfiability Modulo Theories

or

How to build your own Symbolic Execution Tool in 90 Minutes

Philipp Rümmer
February 17, 2021
Outline

- A very short overview of SAT and SMT
- Tutorial: how to apply an SMT solver to implement symbolic execution
Propositional SATisfiability
Propositional logic

- Defined by grammar:

\[ \phi ::= x \mid \phi \land \phi \mid \phi \lor \phi \mid \neg \phi \]

where \( x \) is a proposition or Boolean variable

- Further operators can be defined, e.g.:

\[ \rightarrow, \leftrightarrow \]

Negation, often written \( \neg \phi \)
Some important notions

Definition (Satisfiable, Unsatisfiable, Valid)
A formula $C$ over $n$ variables is *satisfiable* if there is an assignment of the variables that makes $C$ evaluate to *true*; $C$ is *unsatisfiable* if it evaluates to *false* for every assignment; $C$ is *valid* (a tautology) if it evaluates to *true* for every assignment.

$(x \lor \overline{y}) \land (y \lor z)$  
\{ $x \mapsto true, z \mapsto true$ \}  
\text{Sat}

$(x \rightarrow (y \rightarrow z)) \rightarrow ((x \rightarrow y) \rightarrow (x \rightarrow z))$  
\text{Valid}
The SAT Problem

Def.: SAT Solver
Input: Propositional formula $C$ in $n$ variables
Output: $C$ sat + satisfying assignment (model)
$C$ unsat [ + Proof]

- Canonical NP-complete problem [Cook, 1971]
Progress in SAT

Results of the SAT competition/race winners on the SAT 2009 application benchmarks, 20mn timeout

CPU Time (in seconds)

Number of problems solved
Progress in SAT

The SAT Revolution

Results of the SAT competition/race winners on the SAT 2009 application benchmarks, 20mn timeout
Applications of SAT

Diagram with labeled components and diagrams of DNA structures.
Analysing programs ...

- Just Boolean operations are insufficient, we also need datatypes/-structures
- Arithmetic, functions, arrays, etc.

(and possibly quantifiers)
Theories

- Define a library of common operations, reuse it everywhere

**Definition (theory)**
A (first-order) theory $T$ is specified by a signature $\Sigma_T$ of operations (sorts, functions, predicates), and a class $S_T$ of intended interpretations of the symbols in $\Sigma_T$. 
Define a library of common operations, reuse it everywhere

**Definition (theory)**
A (first-order) theory $T$ is specified by a signature $\Sigma_T$ of operations (sorts, functions, predicates), and a class $S_T$ of intended interpretations of the symbols in $\Sigma_T$.

Resulting notion: Satisfiability modulo $T \rightarrow \text{SMT}$
Satisfiability Modulo Theories
Def.: SAT Solver
Input: Propositional formula $C$ in $n$ variables
Output: $C$ sat + satisfying assignment (model)
$C$ unsat [+ Proof]

Def.: SAT Modulo Theories Solver
Input: First-order formula $C$ in $n$ variables and theories $T_1, \ldots, T_m$
Output: $C$ sat + satisfying assignment (model)
$C$ unsat [+ Proof]
Applications of SMT

\[(x > 3.0 \lor y < 2.0) \land (x = y \lor x \neq y - 1.0) \land y < 1.0\]

\[\begin{align*}
i &= 0; \\
x &= j; \\
\text{while } (i < 50) \{ \\
&\quad i++; \\
&\quad x++; \\
\} \\
\text{if } (j == 0) \\
&\quad \text{assert } (x \geq 50);\
\end{align*}\]
Applications of SMT

\[(\lambda \beta) \land (\forall a \land (\exists b \land (c \land d))) \land (e \lor f)\]

\[
i = 0;
\] \[
x = j;
\]
\[\text{while} \ (i < 50) \{\]
\[\quad i++;
\] \[
\quad x++;
\] \[
\}\]
\[\text{if} \ (j == 0) \]
\[\quad \text{assert} \ (x >= 50);\]
High-level DPLL(T)

Formula \rightarrow SAT Solver

Satisfying assignment

SAT Solver
Boolean Skeleton

Theory Solvers

Conflict clauses

UNSAT \rightarrow SAT
Satisfiability Modulo Theories in Practice
Example: LRA/LIA

Linear Rational Arithmetic

Linear Integer Arithmetic

\[ 2y - z > 2 \vee x = 1 \]
\[ 3x - z > 6 \vee x \neq 1 \]
\[ 2z - 4y > 5 \vee x = 1 \]
\[ y - x \not> 6 \vee x \neq 1 \]
In SMT-LIB

=set-logic QF_LIA

(declare-const x Int)
(declare-const y Int)
(declare-const z Int)

(assert (or (> (- (* 2 y) z) 2) (= x 1)))
(assert (or (> (- (* 3 x) z) 6) (not (= x 1))))
(assert (or (> (- (* 2 z) (* 4 y)) 5) (= x 1)))
(assert (or (not (> (- y z) 6)) (not (= x 1))))

(check-sat)
(get-model)
Most Important Commands

- Declaration of variables/symbols:
  
  declare-const
  declare-fun

- Assertion of constraints/formulas:
  
  assert

- Checking satisfiability of assertions:
  
  check-sat
  get-model
Some Common Theories

- LIA, NIA: integer arithmetic
- LRA, NRA: real arithmetic
- FP: floating-point arithmetic
- BV: bitvectors
- EUF: equality + uninterpretable functions
- Arrays
- ADTs: algebraic data-types
- Strings
Some Known SMT solvers

- Z3
- CVC4
- MathSAT
- Yices
- OpenSMT
- Boolector, Bitwuzla
- SMTInterpol
- UU: Norn, TRAU, Sloth, Ostrich; Princess
SMT for Symbolic Execution
Example

1. void foobar(int a, int b) {
2.     int x = 1, y = 0;
3.     if (a != 0) {
4.         y = 3+x;
5.     } if (b == 0)
6.         x = 2*(a+b);
7.     }
8.     assert(x-y != 0);
9. }

The goal is to find the inputs that make the assertion fail.

- Random testing with concrete values unlikely generate the inputs.
- Symbolic execution overcomes the limitation of random testing by reasoning on classes of inputs, rather than single input values.
Symbolic Execution Tree

A. \( \sigma = \{ a \mapsto \alpha_a, b \mapsto \alpha_b \} \) \( \pi = \text{true} \)
2. \text{int } x = 1, \ y = 0

\( \alpha_a \neq 0 \)

B. \( \sigma = \{ a \mapsto \alpha_a, b \mapsto \alpha_b, x \mapsto 1, y \mapsto 0 \} \) \( \pi = \text{true} \)
3. \text{if } (a != 0)

\( \alpha_a = 0 \)

\( \alpha_b = 0 \)

C. \( \sigma = \{ a \mapsto \alpha_a, b \mapsto \alpha_b, x \mapsto 1, y \mapsto 0 \} \) \( \pi = \alpha_a \neq 0 \)
4. \( y = 3 + x \)

D. \( \sigma = \{ a \mapsto \alpha_a, b \mapsto \alpha_b, x \mapsto 1, y \mapsto 0 \} \) \( \pi = \alpha_a = 0 \)
8. \text{assert}(x - y != 0)

\( 1 - 0 = 0 \land \alpha_a = 0 \iff \text{false} \) OK

E. \( \sigma = \{ a \mapsto \alpha_a, b \mapsto \alpha_b, x \mapsto 1, y \mapsto 4 \} \) \( \pi = \alpha_a \neq 0 \)
5. \text{if } (b == 0)

\( \alpha_b \neq 0 \)

F. \( \sigma = \{ a \mapsto \alpha_a, b \mapsto \alpha_b, x \mapsto 1, y \mapsto 4 \} \) \( \pi = \alpha_a \neq 0 \land \alpha_b = 0 \)
6. \( x = 2 \cdot (a+b) \)

\( \sigma = \{ a \mapsto \alpha_a, b \mapsto \alpha_b, x \mapsto 2(\alpha_a + \alpha_b), y \mapsto 4 \} \) \( \pi = \alpha_a \neq 0 \land \alpha_b = 0 \)
8. \text{assert}(x - y != 0)

G. \( \sigma = \{ a \mapsto \alpha_a, b \mapsto \alpha_b, x \mapsto 1, y \mapsto 4 \} \) \( \pi = \alpha_a \neq 0 \land \alpha_b \neq 0 \)
8. \text{assert}(x - y != 0)

\( 1 - 4 = 0 \land \alpha_a \neq 0 \land \alpha_b \neq 0 \iff \text{false} \) OK

H. \( \sigma = \{ a \mapsto \alpha_a, b \mapsto \alpha_b, x \mapsto 2(\alpha_a + \alpha_b), y \mapsto 4 \} \) \( \pi = \alpha_a \neq 0 \land \alpha_b = 0 \)
8. \text{assert}(x - y != 0)

\( 2(\alpha_a + \alpha_b) - 4 = 0 \land \alpha_a \neq 0 \land \alpha_b = 0 \) if \( \alpha_a = 2 \land \alpha_b = 0 \) ERROR
Typical Architecture

Model checker, SymEx, etc.

Stream of queries

SAT/SMT solver

Answers (models, proofs)
(1) SMT interface layer
(2) ASTs to represent expressions/program
(3) Encoders from expressions to SMT
(4) Depth-first traversal of programs
\( \text{Expr} ::= \ 0 \mid 1 \mid 2 \mid \ldots \mid \mathbf{1} \ldots \)
\( \mid x \quad x \in X \)
\( \mid \text{Expr} \ op \ \text{Expr} \quad \text{op} \in \{+, \ast\} \)

\( \text{BExpr} ::= \ \text{Expr} = \ \text{Expr} \)
\( \mid \text{Expr} \leq \ \text{Expr} \)
\( \mid \neg \ \text{BExpr} \)
\( \mid \text{BExpr} \ bop \ \text{BExpr} \quad \text{bop} \in \{\land, \lor\} \)

\( \text{Prog} ::= \ \text{Skip} \)
\( \mid x := \ \text{Expr} \)
\( \mid \text{assert} (\ \text{BExpr} ) \)
\( \mid \text{Prog} ; \ \text{Prog} \)
\( \mid \text{if} (\ \text{BExpr} ) \ \text{Prog} \ \text{else} \ \text{Prog} \)
\( \mid \text{while} (\ \text{BExpr} ) \ \text{Prog} \)
SMT-LIB Operators

- (and ...), (or ...), (not ...), (=> ...)
- (= b c)
- (ite (= b c) #b101 #b011)
- (let ((a #b001) (b #b010)) (= a b))
- (exists ((x (_ BitVec 2))) (= #b101 x)) (forall ...)
- (! (= b c) :named X)
SMT-LIB Bit-vector Operators

- (_ BitVec 8)
- #b1010, #xff2a, (_ bv42 32)
- (= (concat #b1010 #b0011) #b10100011)
- (= ((_ extract 1 3) #b10100011) #b010)
- Unary: bvnot, bvneg
- Binary: bvand, bvor, bvadd, bvmul, bvudiv, bvurem, bvshl, bvlshr
- (bvult #b0100 #b0110), (bvsle ...)
- And many more derived operators ...
SMT-LIB Bit-vector Operators

http://smtlib.cs.uiowa.edu/logics-all.shtml#QF_BV

- (_ BitVec 8)
- #b1010, #xff2a, (_ bv42 32)
- (= (concat #b1010 #b0011) #b10100011)
- (= ((_ extract 1 3) #b10100011) #b010)
- **Unary**: bvnot, bvneg
- **Binary**: bvand, bvor, bvadd, bvmul, bvdiv, bvurem, bvshl, bvlshr
- (bvult #b0100 #b0110), (bvsle ...)
- And many more derived operators
The Assertion Stack

- Holds both assertions and declarations, but no options
- Important for incremental use of solver
- \((\text{push } n) \rightarrow \text{add } n \text{ new frames to the stack}\)
- \((\text{pop } n) \rightarrow \text{pop } n \text{ frames from the stack}\)
Outlook

- More material:
  - “Decision procedures” book
    - http://ssa-school-2016.it.uu.se/
    - http://www.sc-square.org/CSA/school/
    - http://ssa-school-2018.cs.manchester.ac.uk/

- Implementation:
  - https://github.com/pruemmer/nano-symex
Thank you for your attention!

Questions?
More SMT-LIB Commands

- `(set-logic QF_LIA)`
- `(set-option ...)`
- `(declare-const b (_ BitVec 8))`
- `(declare-fun f ((x (_ BitVec 2))) Bool)`
- `(assert (= b #b10100011))`
- `(check-sat)`
- `(get-value (b)), (get-model)`
- `(get-unsat-core)`
- `(push 1), (pop 1)`
- `(reset), (exit)`
More SMT-LIB Commands

- (set-logic QF_LIA)
- (set-option ...)
- (declare-const b (_ BitVec 8))
- (declare-fun f ((x (_ BitVec 2))) Bool)
- (assert (= b #b10100011))
- (check-sat)
- (get-value (b)), (get-model)
- (get-unsat-core)
- (push 1), (pop 1)
- (reset), (exit)

Z3, and many solvers don't care ...