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:

$$\phi \ ::= \ x \mid \phi \wedge \phi \mid \phi \vee \phi \mid \bar{\phi}$$

Negation, often written $\neg\phi$

where x is a **proposition** or **Boolean variable**

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

$$\rightarrow, \leftrightarrow$$

Some important notions

$$(x \vee \bar{y}) \wedge (y \vee z)$$

Sat

$$\{x \mapsto true, z \mapsto true\}$$

$$(x \to (y \to z)) \to ((x \to y) \to (x \to z))$$

Valid

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* it evaluates to *false* for every assignment;
C is *valid* (a *tautology*) if it evaluates to *true* for every assignment.

The SAT Problem

Def.: SAT Solver

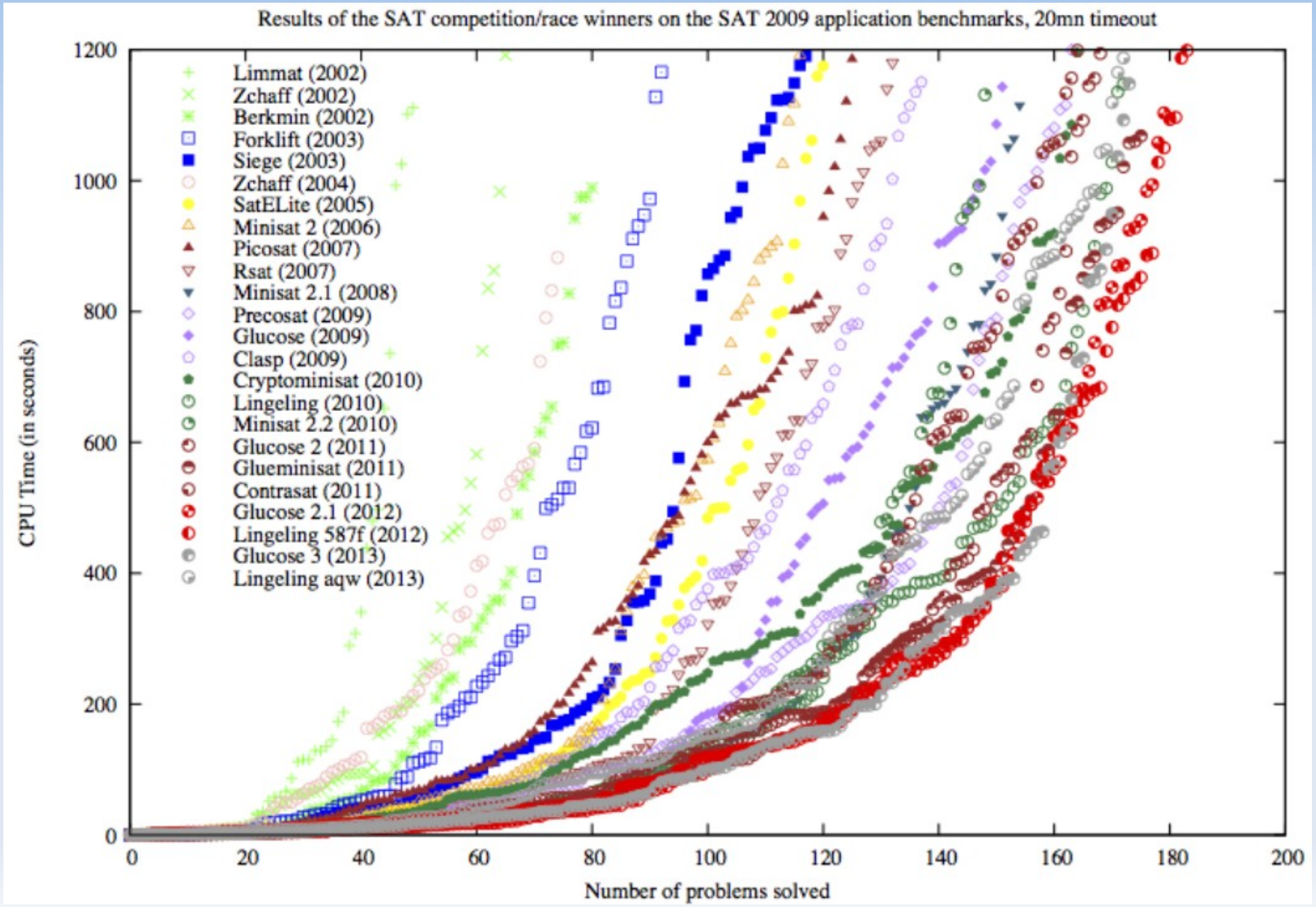
Input: **Propositional** formula *C* in *n* variables

Output: ***C* sat** + satisfying assignment (model)

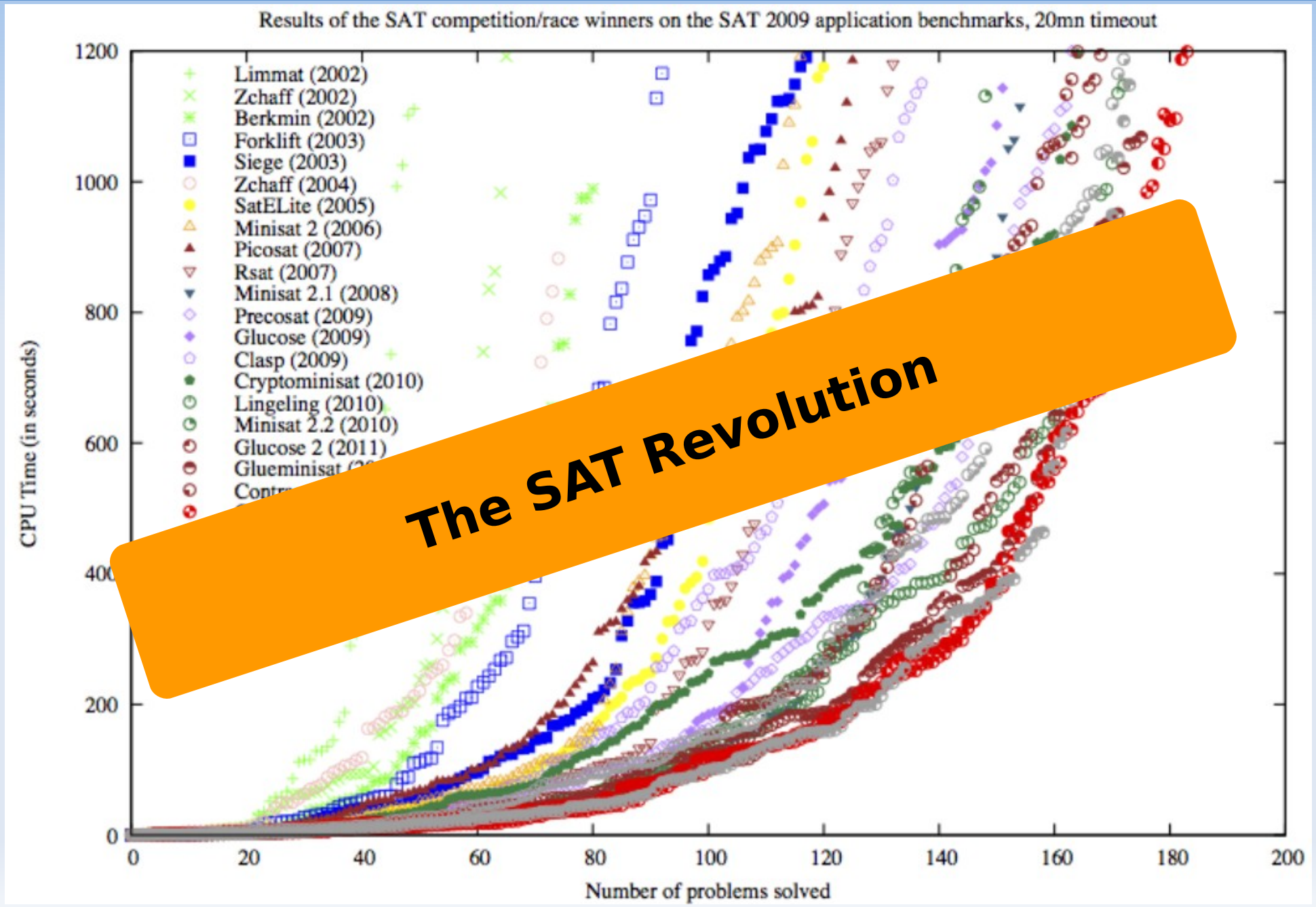
***C* unsat** [+ Proof]

- Canonical NP-complete problem
  [Cook, 1971]

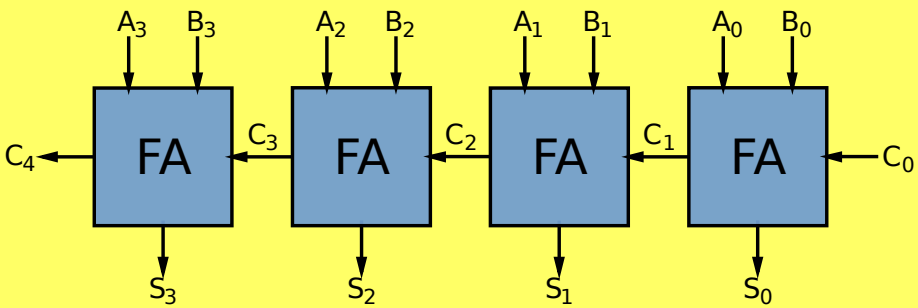
Progress in SAT

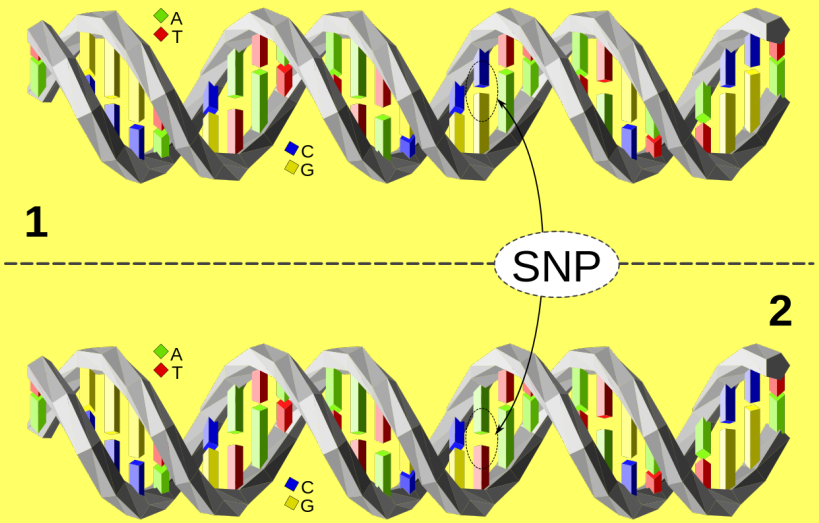


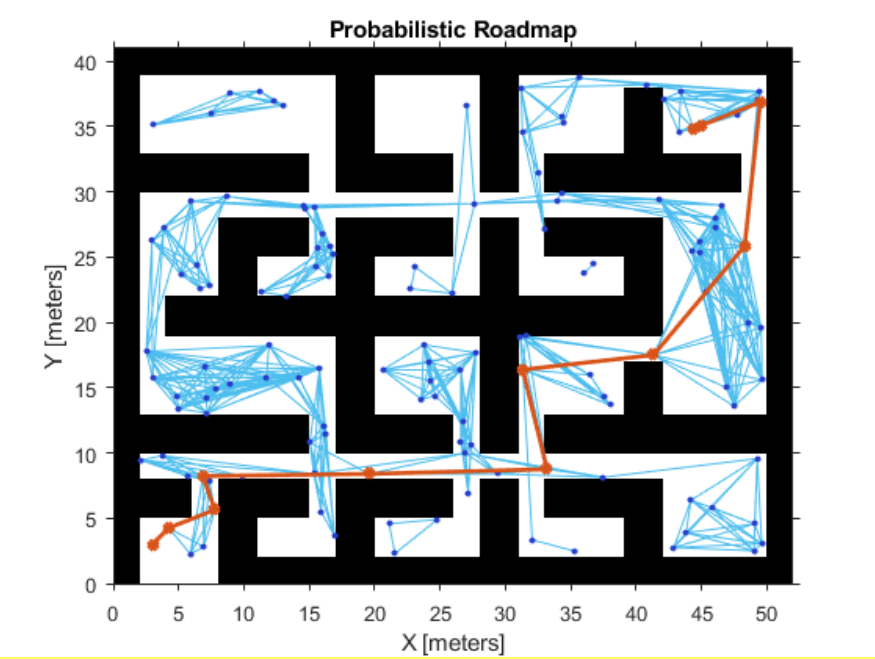
Progress in SAT



Applications of SAT







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 Σ_T of operations (sorts, functions, predicates), and a class $\mathcal{S}_T$ of intended interpretations of the symbols in Σ_T.

Theories

- Define a library of common operations, reuse it everywhere

Definition (theory)
A (first-order) theory T is specified by a signature Σ_T of operations (sorts, functions, predicates), and a class $\mathcal{S}_T$ of intended interpretations of the symbols in Σ_T.

- Resulting notion:
  Satisfiability modulo *T* → **SMT**

Satisfiability Modulo Theories

SAT and SMT

Def.: SAT Solver

Input: **Propositional** formula C
in n variables

Output: ***C* sat** + satisfying assignment (model)
***C* unsat** [+ Proof]

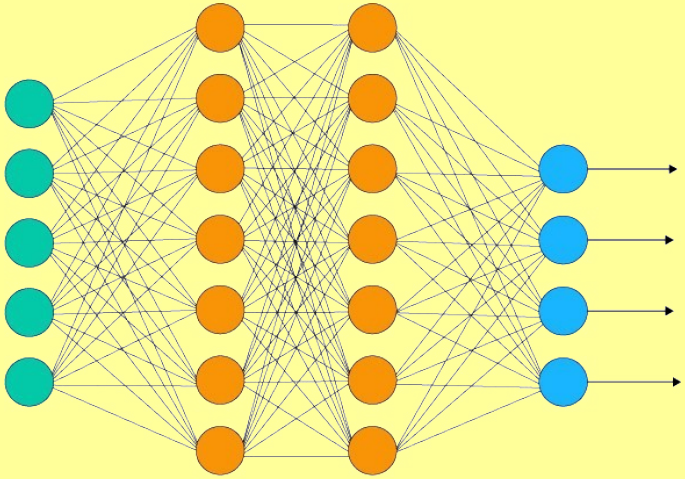
Def.: SAT Modulo Theories Solver

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

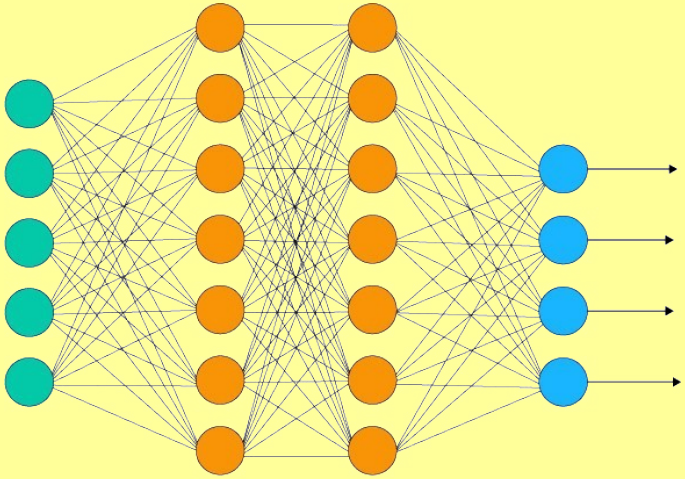
$$(x > 3.0 \vee y < 2.0) \wedge$$
$$(x = y \vee x \neq y - 1.0) \wedge$$
$$y < 1.0$$



```
i = 0;
x = j;
while (i < 50) {
    i++;
    x++;
}
if (j == 0)
    assert (x >= 50);
```

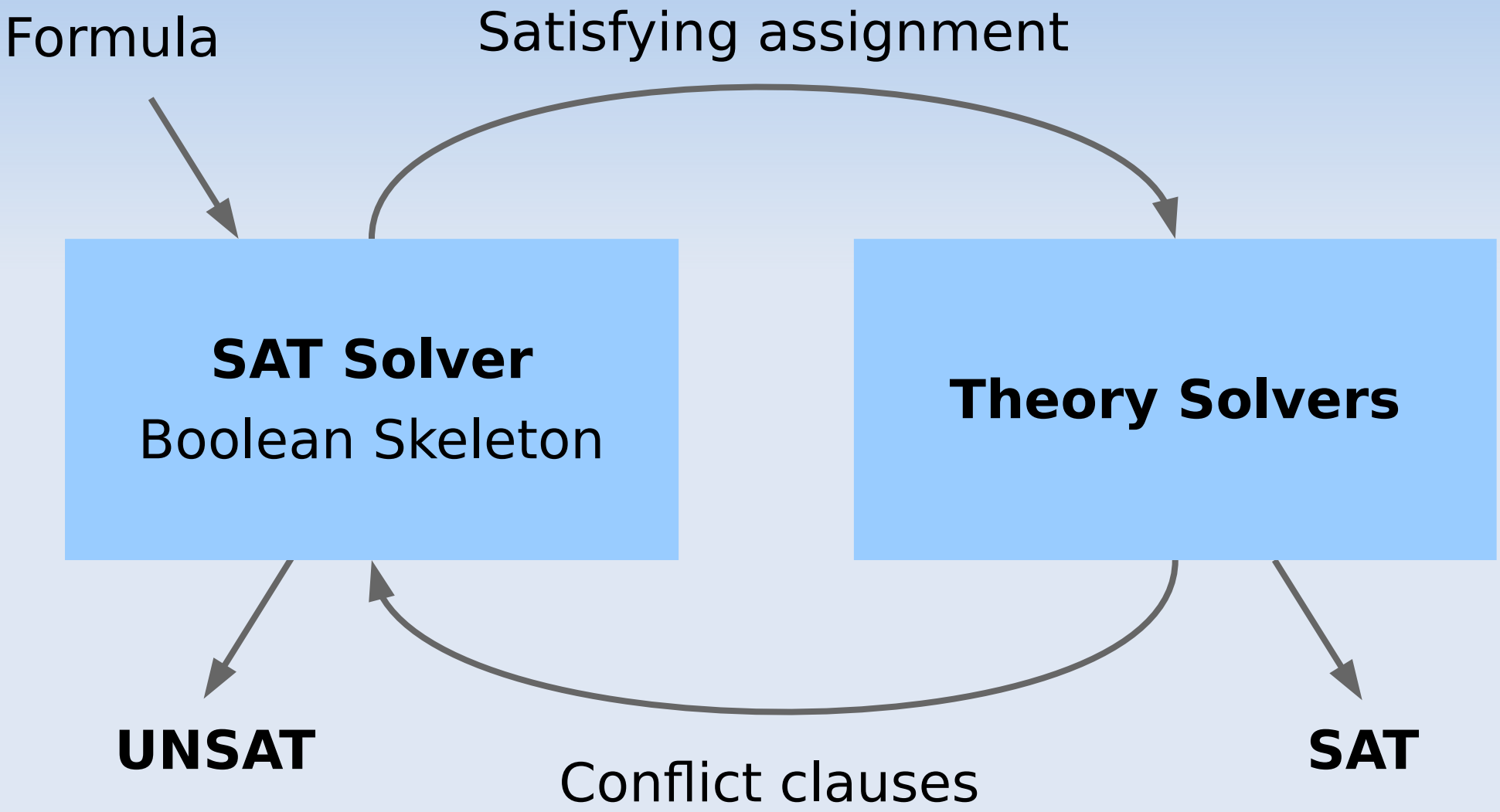
Applications of SMT

(▇▇▇▇▇▇ ∨ ▇▇▇▇▇▇) ∧
(▇▇▇▇ ∨ ▇▇▇▇▇▇▇▇▇) ∧
▇▇▇▇▇▇



```
i = 0;
x = j;
while (i < 50) {
    i++;
    x++;
}
if (j == 0)
    assert (x >= 50);
```

High-level DPLL(T)



Satisfiability Modulo Theories in Practice

Example: LRA/LIA

Linear Rational Arithmetic

Linear Integer Arithmetic

$$
\begin{aligned}
2y - z &> 2 \lor x = 1 \\
3x - z &> 6 \lor x \neq 1 \\
2z - 4y &> 5 \lor x = 1 \\
y - x &\not> 6 \lor x \neq 1
\end{aligned}
$$

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)
```

Permalink: https://rise4fun.com/Z3/E3bF

The SMT-LIB Standard
Version 2.6

Clark Barrett Pascal Fontaine Cesare Tinelli

Release: 2017-07-18

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 + uninterpr. 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

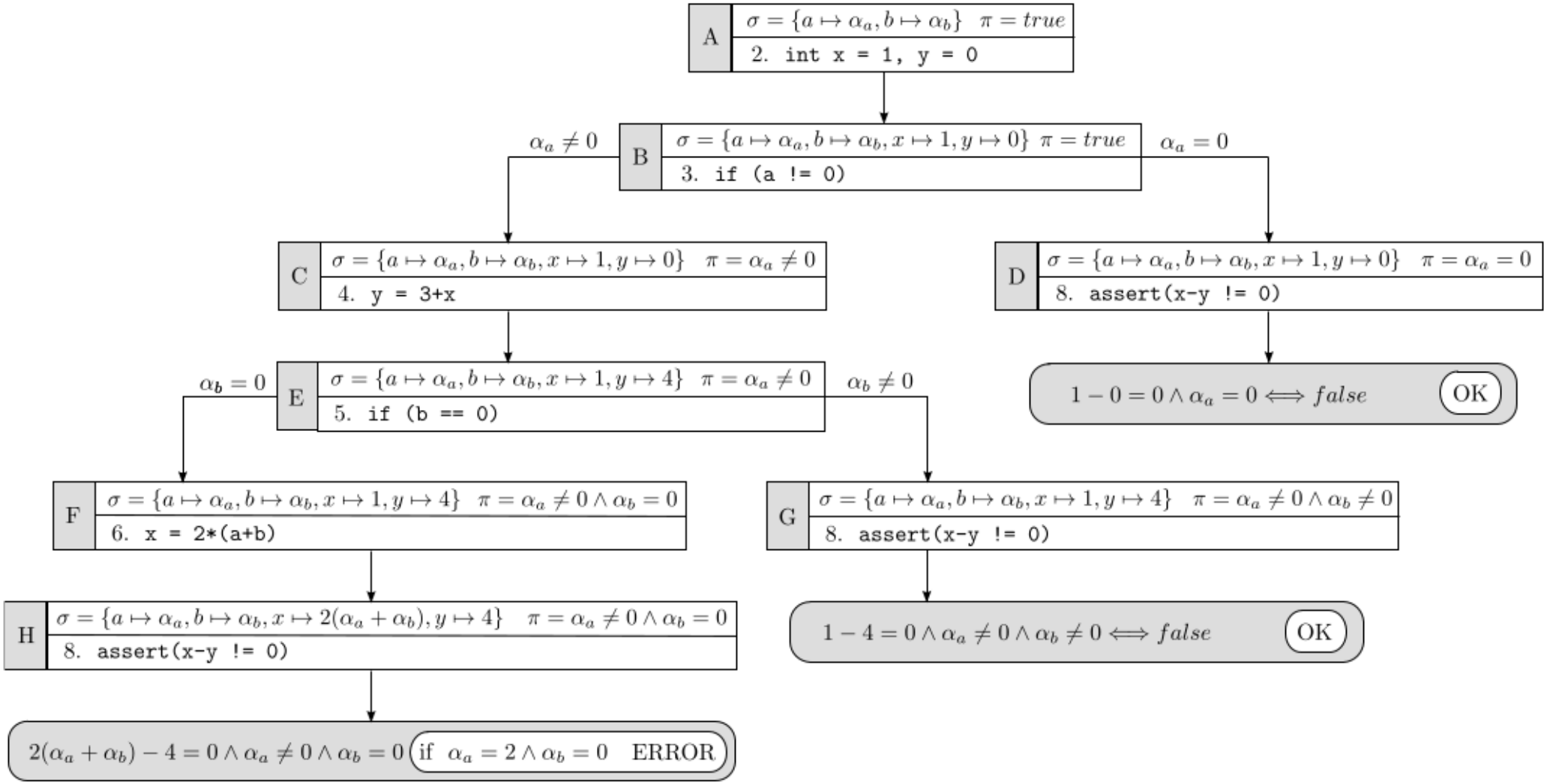
Example

```
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.  }
```

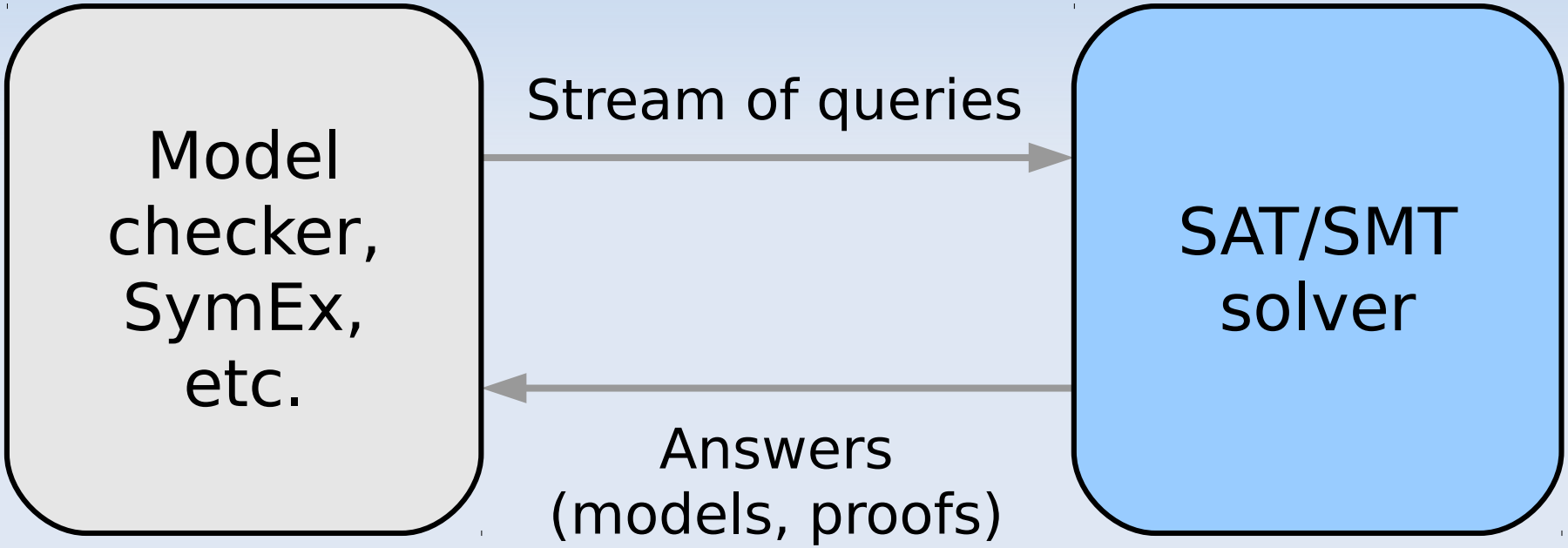
The goal is to find the inputs that make the assertion fail.

- 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



Typical Architecture



The Master Plan

(1) SMT interface layer

(2) ASTs to represent expressions/program

(3) Encoders from expressions to SMT

(4) Depth-first traversal of programs

$$
\begin{aligned}
\mathrm{Expr} &::= 0 \mid 1 \mid 2 \mid \cdots \mid -1 \mid \cdots \\
&\mid x && x \in X \\
&\mid \mathrm{Expr}\ \mathrm{op}\ \mathrm{Expr} && \mathrm{op} \in \{+, *\} \\
\mathrm{BExpr} &::= \mathrm{Expr} = \mathrm{Expr} \\
&\mid \mathrm{Expr} \leqslant \mathrm{Expr} \\
&\mid \neg\ \mathrm{BExpr} \\
&\mid \mathrm{BExpr}\ \mathrm{bop}\ \mathrm{BExpr} && \mathrm{bop} \in \{\wedge, \vee\} \\
\mathrm{Prog} &::= \mathrm{Skip} \\
&\mid x := \mathrm{Expr} \\
&\mid \mathrm{assert}\ (\mathrm{BExpr}) \\
&\mid \mathrm{Prog}\ ;\ \mathrm{Prog} \\
&\mid \mathrm{if}\ (\mathrm{BExpr})\ \mathrm{Prog}\ \mathrm{else}\ \mathrm{Prog} \\
&\mid \mathrm{while}\ (\mathrm{BExpr})\ \mathrm{Prog}
\end{aligned}
$$

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

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, 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, bvudiv, bvurem, bvshl, bvlshr`
- `(bvult #b0100 #b0110), (bvsle ...)`
- *And many more derived operators*

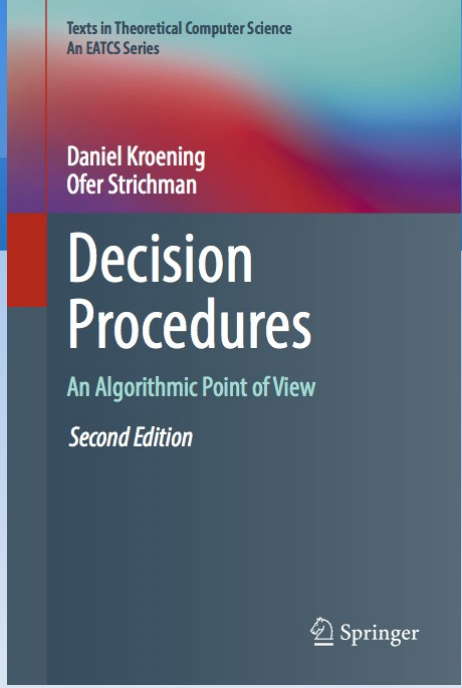
Unsigned

Signed

The Assertion Stack

- Holds both assertions and declarations, but no options
- Important for incremental use of solver
- (push *n*) → add *n* new frames to the stack
- (pop *n*) → pop *n* frames from the stack

Outlook

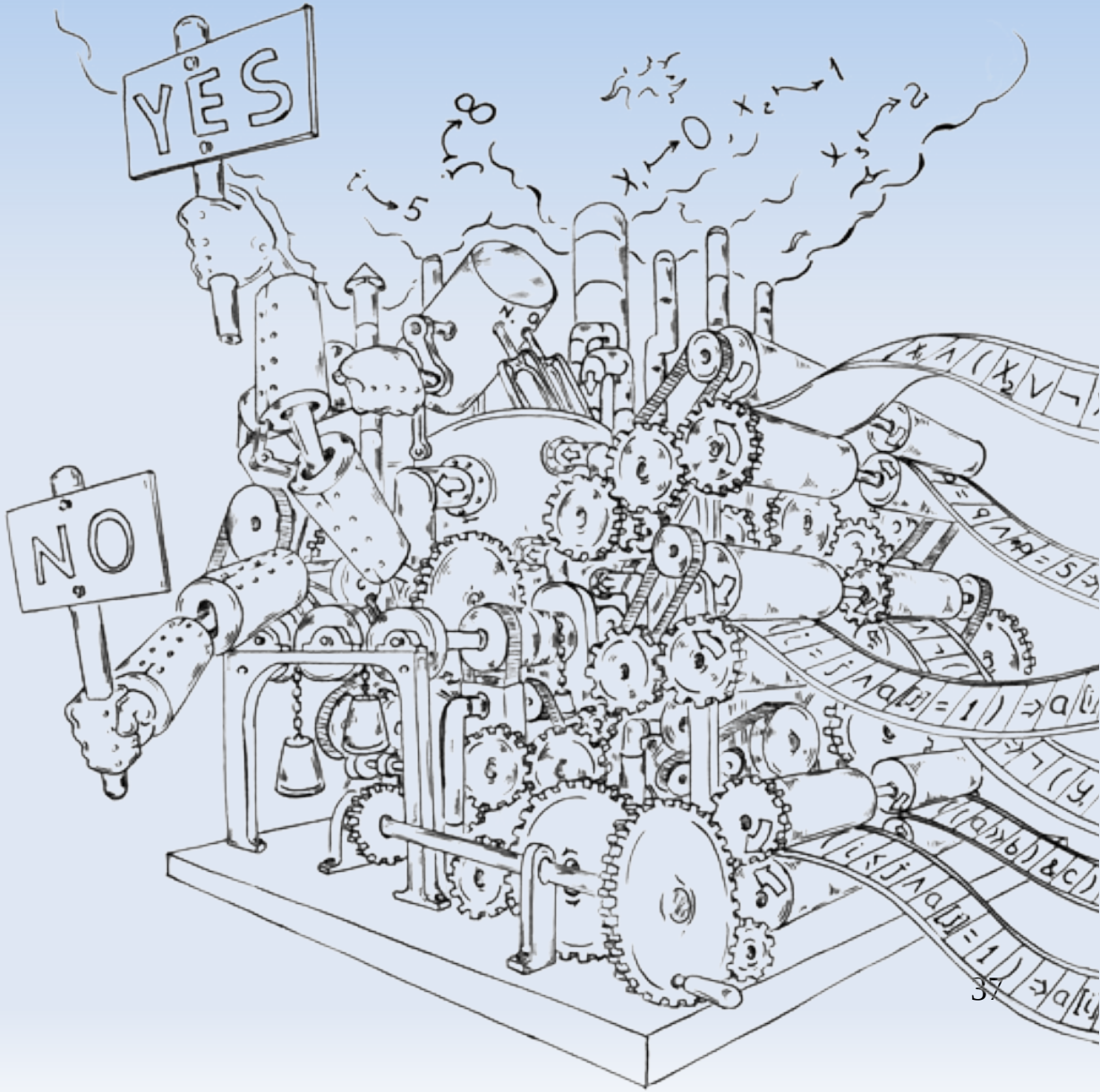


- More material:
  - “Decision procedures” book
  - http://ssa-school-2016.it.uu.se/
  - http://www.sc-square.org/CSA/school/
  - http://ssa-school-2018.cs.manchester.ac.uk/
  - https://alexeyignatiev.github.io/ssa-school-2019/

- 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 …