From SAT to SMT

The Next Frontier in Problem Solving and Formal Verification

Konstantin Chukharev

November 2024

ITMO University

Konstantin Chukharev

Lipen Lipenx kchukharev kchukharev@itmo.ru

PhD student, researcher and lecturer at ITMO University.
Senior academic consultant at Huawei R&D Toolchain Lab.

Received Bachelor's degree in Robotics and Master's degree in Computer Science from ITMO University. Finished one-year program in Bioinformatics Insitute, Saint Petersburg.

Research interests: SAT, formal methods, software verification, automata synthesis, model checking, Rust.

From SAT to SMT

The Next Frontier in Problem Solving and Formal Verification.

Abstract: This talk introduces the transition from the classical Boolean Satisfiability Problem (SAT) to the more expressive Satisfiability Modulo Theories (SMT). We explore the motivations behind SMT, and the key theories that extend the capabilities of SAT solvers. The talk also covers the architecture of SMT solvers and their applications in software analysis, in particular, for symbolic execution.

Mathematics is not about numbers, equations, computations, or algorithms: it is about understanding.

— William Paul Thurston

1. Introduction to SAT

SAT is the classical **NP-complete** problem of determining if a given **Boolean formula** is satisfiable, that is, if there **exists a model** an assignment of truth values to the variables that makes the formula true, or **proving** that no such assignment exists.

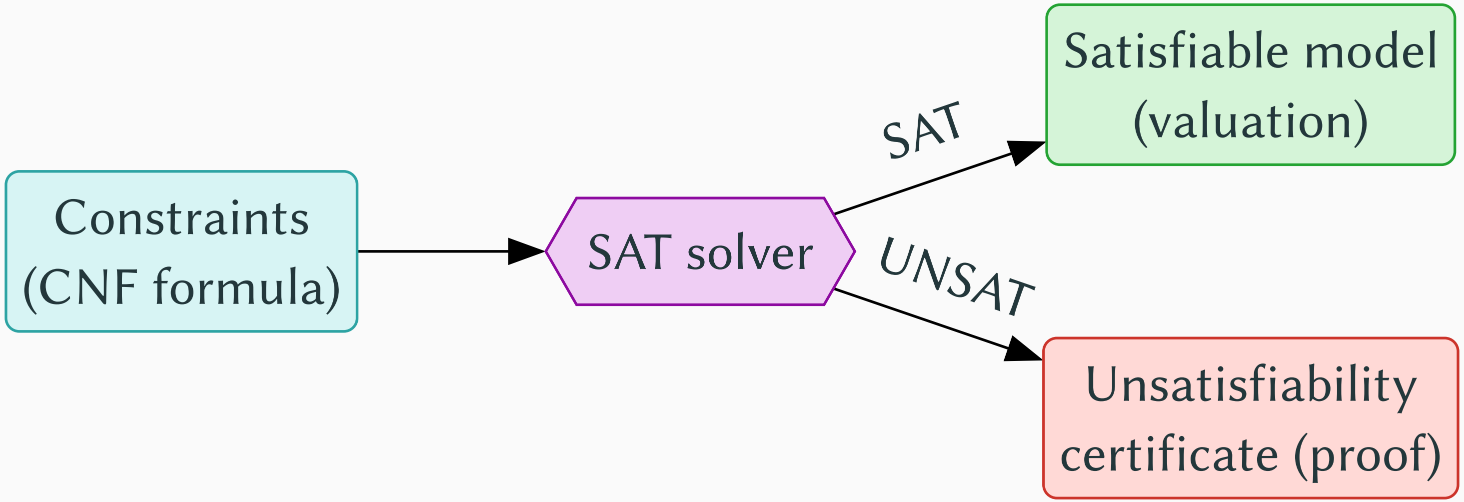
$$\exists X . F(X) = 1$$

Example:

$(\texttt{TIE} \rightarrow \texttt{SHIRT}) \land (\texttt{TIE} \lor \texttt{SHIRT}) \land \neg(\texttt{TIE} \land \texttt{SHIRT}) \land \texttt{TIE}$

Limitations of SAT:

- Restricted to propositional variables.
- Most SAT solvers are limited to CNF formulas.
- Cannot handle arithmetic expressions (e.g., $x + y > 5$) natively.
- Lacks support for data structures like arrays or lists.

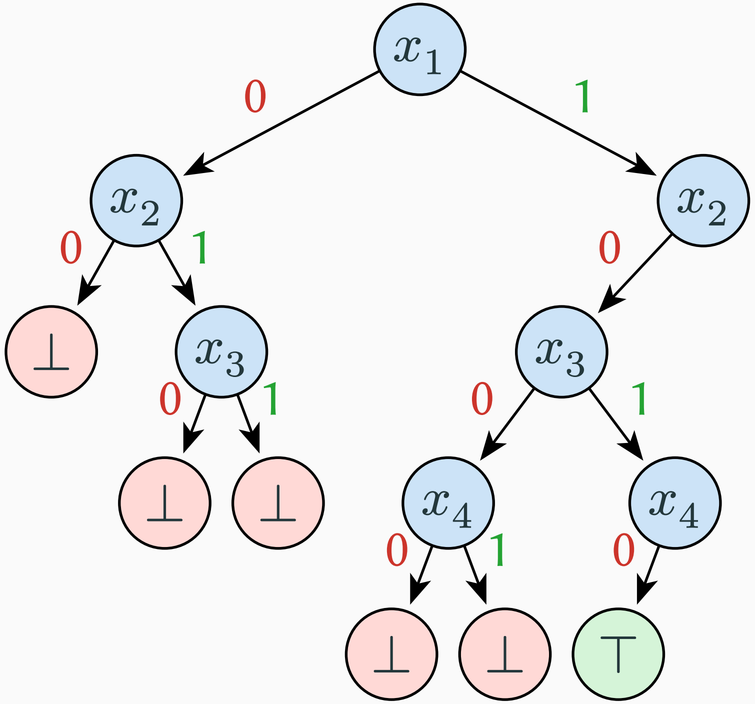


To interact with SAT solvers from Java/Kotlin, use `kotlin-satlib` library.

To interact with SAT solvers from Rust, use `sat-nexus` library.

CDCL Algorithm

```
while true do
    while propagate_gives_conflict() do
        if decision_level == 0 then return UNSAT
        else analyze_conflict()
    end
    restart_if_applicable()
    remove_lemmas_if_applicable()
    if not decide() then return SAT
end
```



2. From SAT to SMT

2.1 Satisfiability Modulo Theories (SMT)

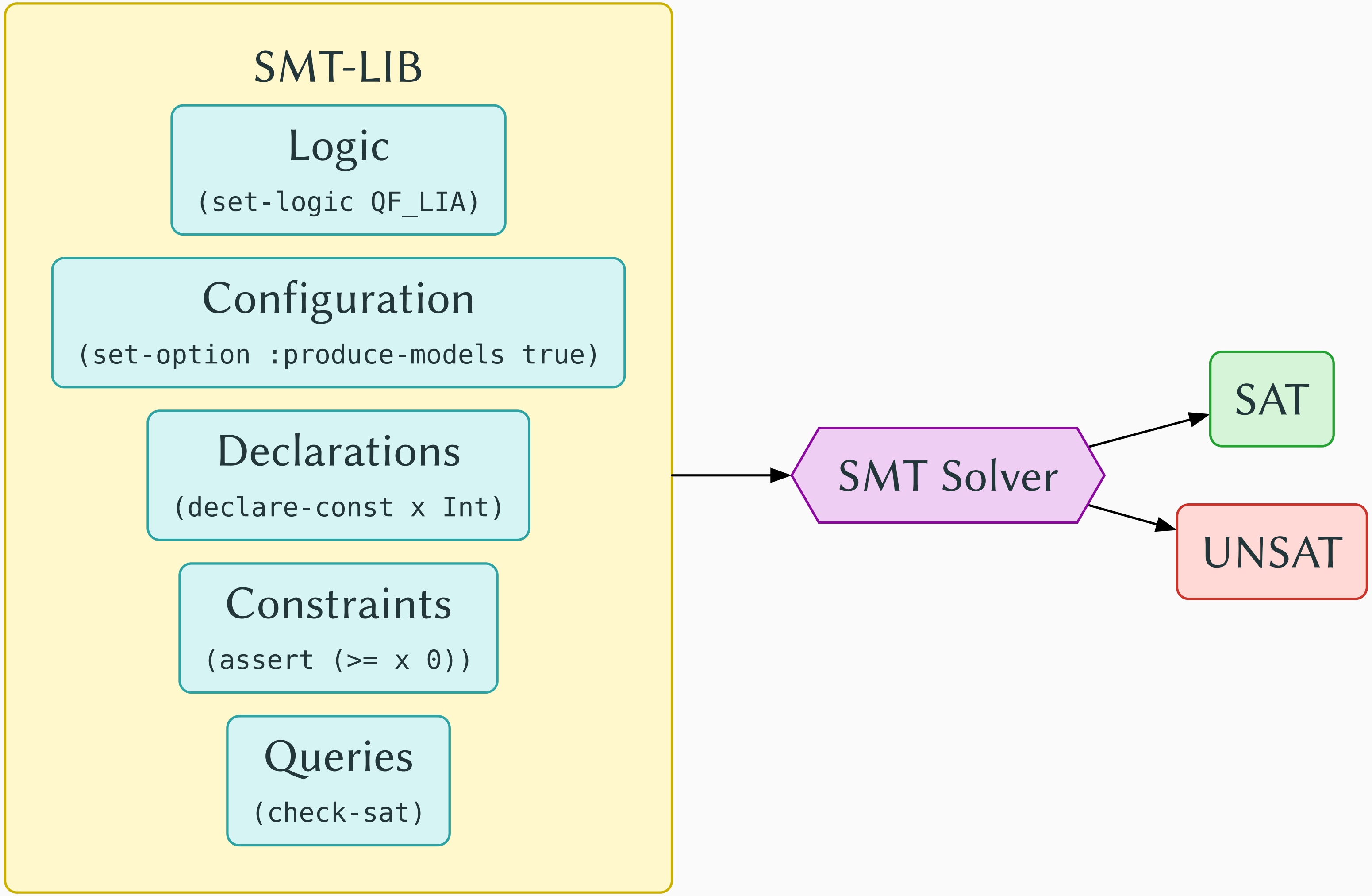
SMT = SAT + Theories.

A **theory** is a set of logical formulas modeling a particular domain.

Common components of theories:

- **Logic**: Propositional and first-order logic.
- **Arithmetic**: Numbers, math operations, inequalities.
- **Arrays**: Access (`read`) and update (`write`) operations.
- **Bit-Vectors**: Bitwise operations on fixed-size (e.g. 32-bit) binary representations.
- **Uninterpreted Functions**: Functions without a fixed interpretation.

Logic	Arithmetic	Arrays	Bit-Vectors	Uninterpreted Functions
$\forall x\, \exists y.(x \to y)$	$2x + 3 \geq y$	$A[i] = x$	$x_{32} = y_{32} \oplus z_{32}$	$f(x) = y$


SMT-LIB
Logic
(set-logic QF_LIA)
Configuration
(set-option :produce-models true)
Declarations
(declare-const x Int)
Constraints
(assert (>= x 0))
Queries
(check-sat)
SMT Solver
SAT
UNSAT

The basis of almost all "practical" SMT theories is a classical **first-order logic with equality**.

- **Variables**: Boolean ($x, y, z \in \mathbb{B}$) or from some domain $\mathbb{D}$ (numbers, objects, ...).
- **Logical connectives**: $\wedge$, $\vee$, $\neg$, $\rightarrow$, $\leftrightarrow$.
- **Quantifiers**: universal ($\forall$) and existential ($\exists$).
- **Functions** and **Predicates**: $f : \mathbb{D} \rightarrow \mathbb{D}$ and $P : \mathbb{D} \rightarrow \mathbb{B}$.
- **Equality**: "$=$" is a binary relation symbol with the following axioms:
  - Reflexivity: $\forall x.(x = x)$.
  - Substitution for functions: $\forall x, y.(x = y) \rightarrow (f(..., x, ...) = f(..., y, ...))$.
  - Substitution for formulas: $\forall x, y.(x = y) \rightarrow (\varphi(x) \rightarrow \varphi(y))$.

Examples: $\forall x\, \exists y.(x \rightarrow y)$ $\quad \exists x.P(x)$ $\quad \forall x \forall y.P(f(x)) \rightarrow \neg(P(x) \rightarrow Q(f(y), x, z))$

- "$=$" is **equality**, f is an **uninterpreted function**.
- If the background logic is **FOL with equality**, then EUF is **empty** theory.
  - **Example** (UNSAT formula):

$$a \cdot (f(b) + f(c)) = d \quad \land \quad b \cdot (f(a) + f(c)) \neq d \quad \land \quad (a = b)$$

  - Abstracted formula (still UNSAT):

$$h(a, g(f(b), f(c))) = d \quad \land \quad h(b, g(f(a), f(c))) \neq d \quad \land \quad (a = b)$$

  - Both formulas are **unsatisfiable**, without any arithmetic reasoning.
- EUF is used to abstract "non-supported constructions", e.g. non-linear multiplication.

2.6 SMT Theories: Arithmetics (Ints, Reals, Floating Point)

Restricted fragments support more efficient methods.

Logic	**Example expression**
Linear arithmetic (LIA, LRA)	$x + 2y \leq 5$
Non-linear arithmetic (NIA, NRA)	$2xy + 4xz^2 - 5y \leq 10$
Difference logic (DL)	$x - y \bowtie 3,\ \text{where} \bowtie \in \{<, >, \leq, >, =\}$
UTVPI (Unit Two-Variable Per Inequality)	$ax + by \bowtie d,\ \text{where } a, b \in \{-1, 0, 1\}$

Commonly, variables are **Reals** or **Integers**. But there are also:

- Floating-point arithmetic (IEEE 754 standard).
- Rational arithmetic.

Theory of arrays defines two "interpreted" functions: **`read`** and **`write`**.

Axioms:

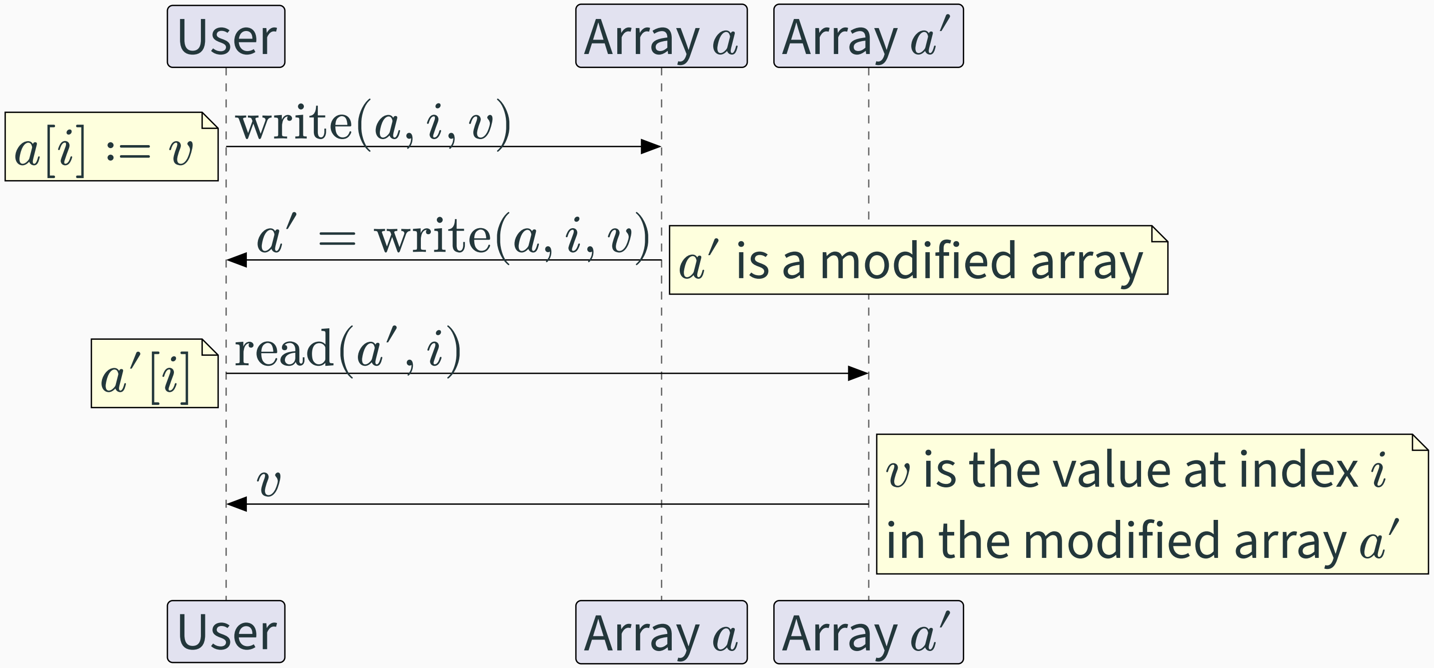
- $\forall a \forall i \forall v. (\mathrm{read}(\mathrm{write}(a, i, v), i) = v)$
- $\forall a \forall i \forall j \forall v. (i \neq j) \rightarrow (\mathrm{read}(\mathrm{write}(a, i, v), j) = \mathrm{read}(a, j))$

Extensionality: $\forall a \forall b \; (\forall i \; (\mathrm{read}(a, i) = \mathrm{read}(b, i))) \rightarrow (a = b)$

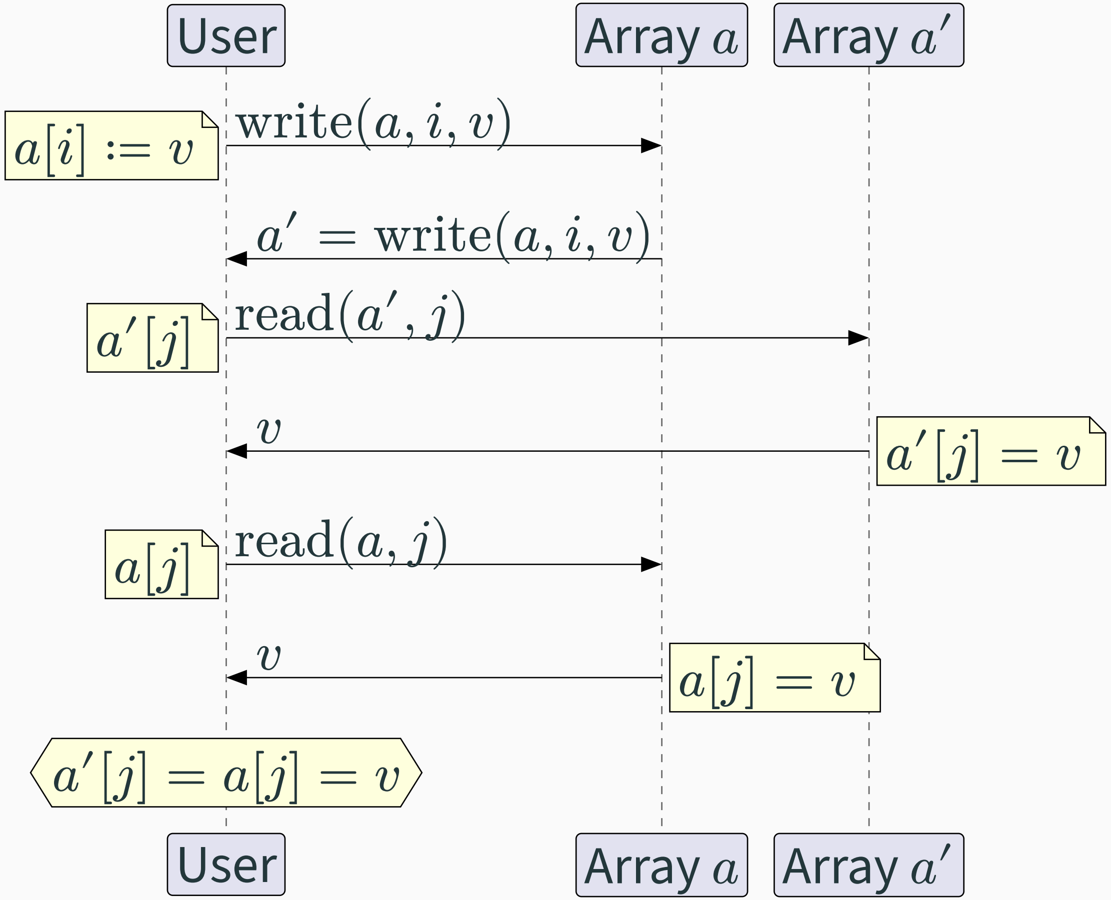
Example:

$$\Gamma = \{\mathrm{write}(a, i, x) \neq b, \mathrm{read}(b, i) = y, \mathrm{read}(\mathrm{write}(b, i, x), j) = y, a = b, i = j\}$$

First axiom: $\forall a \forall i \forall v.(\text{read}(\text{write}(a, i, v), i) = v)$



Second axiom: $\forall a \forall i \forall j \forall v.(i \neq j) \to (\mathrm{read}(\mathrm{write}(a, i, v), j) = \mathrm{read}(a, j))$



Try it online!
https://compsys-tools.ens-lyon.fr/z3
https://jfmc.github.io/z3-play/

```
(set-logic QF_AX) ; Arrays theory

(declare-sort Index)
(declare-sort Element)
(declare-fun a () (Array Index Element))
(declare-fun b () (Array Index Element))
(declare-fun i () Index)
(declare-fun j () Index)
(declare-fun x () Element)
(declare-fun y () Element)

(assert (distinct (store a i x) b))      ; write(a, i, x) != b
(assert (= (select b i) y))              ; read(b, i) = y
(assert (= (select (store b i x) j) y))  ; read(write(b, i, x), j) = y
(assert (= a b))                         ; a = b
(assert (= i j))                         ; i = j

(check-sat)  ; Check satisfiability
(get-model)  ; Get a model if possible
```

$$\Gamma = \{\mathrm{write}(a, i, x) \neq b, \mathrm{read}(b, i) = y, \mathrm{read}(\mathrm{write}(b, i, x), j) = y, a = b, i = j\}$$

Bit-vector is a vector of bits of some fixed size.

"Numbers" (integers) are represented in binary form as bit-vectors.

Operations on bit-vectors:

- String-like: `concat` and `extract`.
- Logical: `bvnot`, `bvor`, `bvand`, `bvxor`, …
- Arithmetic: `bvadd`, `bvsub`, `bvmul`, `bvudiv`, `bvurem`, …
- Comparisons, shifts, rotations, …

Example: (assume all bit-vectors are of size 3)

$$a[0..1] \neq b[0..1] \quad \land \quad (a \mid b) = c \quad \land \quad c[0] = 0 \quad \land \quad a[1] + b[1] = 0$$

2.12 Modern SMT Solvers

Solver	Distinctive feature
Z3	Supports many theories
CVC5	Academic cutting-edge
Yices	Ultra fast
Bitwuzla	Top choice for bit-vectors
MathSAT	Combination of theories
Alt-Ergo	Deductive reasoning

To interact with SMT solvers from Java/Kotlin, use `KSMT` library.

3. Advanced Topics in SMT

3.1 Solving SMT

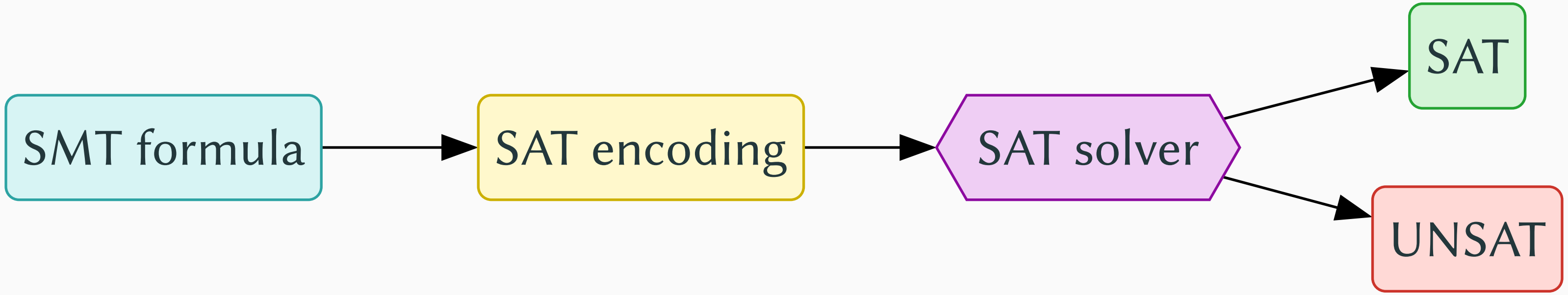
There are **two** main approaches to solving SMT:

- **Eager** approach 🦅

  Encode SMT as SAT and solve using a SAT solver.

- **Lazy** approach 🦥

  Use SAT solver for Boolean part and theory solver for theory-specific parts.



Pros	Cons
✓ Can use the best avaliable SAT solver	✗ Complex encodings for some theories
✓ Simple modular architecture	✗ Scalability issues for large theories
✓ Ideal for finite domains and bounded integers (bit-blasting of bit-vectors)	✗ Unbounded integers and quantifiers can lead to intractable problems

Step 1: Eliminate function and predicate symbols.

Consider a EUF-formula with functions $f(a)$, $f(b)$ and $f(c)$.

- **Ackermann** reduction:
  - Replace each function/predicate with a fresh variable: A, B, C, ...
  - Add clauses: $(a = b) \rightarrow (A = B), \quad (a = c) \rightarrow (A = C), \quad (b = c) \rightarrow (B = C)$
- **Bryant** reduction:
  - Replace $f(a)$ with A
  - Replace $f(b)$ with $\mathrm{ITE}(b = a, A, B)$
  - Replace $f(c)$ with $\mathrm{ITE}(c = a, A, \mathrm{ITE}(c = b, B, C))$
  - Here, ITE stands for "If-Then-Else" conditional expression.

After the first step, atoms in the formula are **equalities** between **constants**.

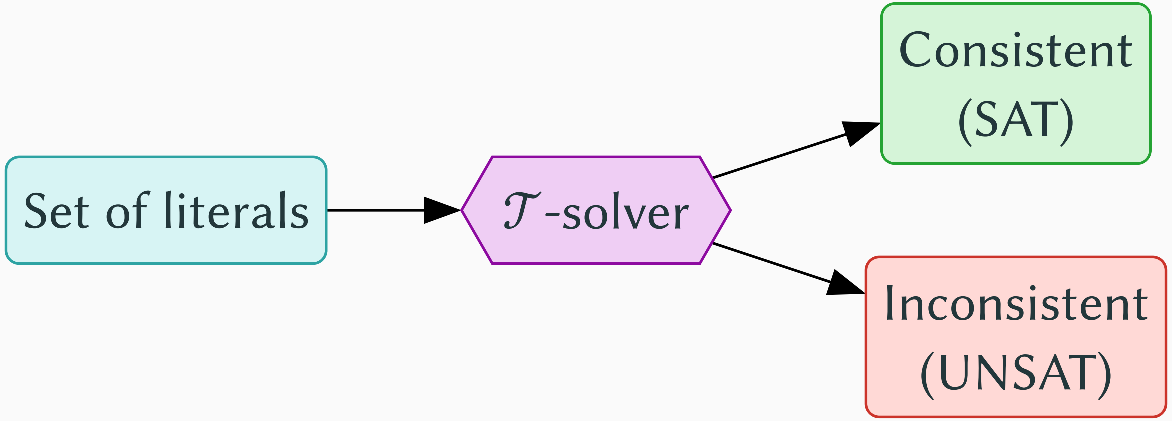
Step 2: Encode into propositional logic.

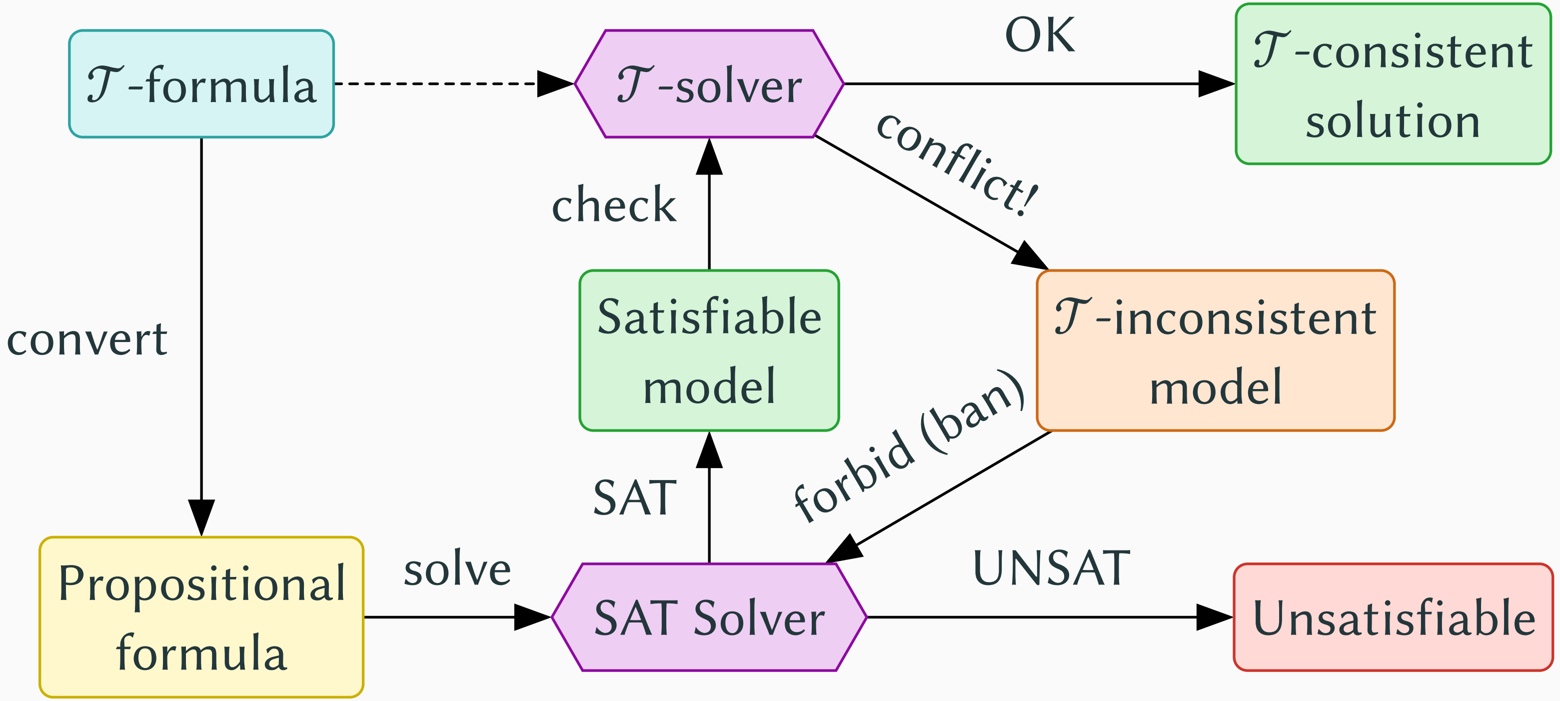
- **Small-domain** encoding:
  - If there are n different constants, there is a model with size at most n.
  - Given n constants, we need $\log n$ bits to represent each constant.
  - Equalities, such as $(a = b)$, are encoded using the bits of a and b.
- **Per-constraint** encoding:
  - Replace each equality $(a = b)$ with a propositional variable $P_{a,b}$.
  - Add transitivity constraints: $P_{a,b} \land P_{b,c} \rightarrow P_{a,c}$.

Done. Throw it into a **SAT solver** and let it do its **magic**!

3.4 Lazy Approach for Solving SMT

“Lazy” means the theory information is used lazily, on demand, when checking the consistency of propositional models found by the SAT solver.


Set of literals
$\mathcal{T}$-solver
Consistent
(SAT)
Inconsistent
(UNSAT)



$\mathcal{T}$-formula
$\mathcal{T}$-solver
OK
$\mathcal{T}$-consistent solution
conflict!
check
convert
Satisfiable model
$\mathcal{T}$-inconsistent model
forbid (ban)
SAT
Propositional formula
solve
SAT Solver
UNSAT
Unsatisfiable

Formulas often involve **multiple** theories, for example:

$$
\begin{aligned}
&(a = b + 2) \land \\
&(A = \mathrm{write}(B, a + 1, 4)) \land \\
&((\mathrm{read}(A, b + 3) = 2) \lor (f(a - 1) \neq f(b + 1)))
\end{aligned}
$$

Here, we have "$+$" from $\mathcal{T}_{\mathrm{LIA}}$, "read" and "write" from $\mathcal{T}_{\mathrm{AX}}$, and "$f(\cdot)$" from $\mathcal{T}_{\mathrm{UF}}$.

Step 1: Purify literals so that each belongs to a single theory.

$$f(f(x) - f(y)) = a \longrightarrow \begin{aligned} f(e_1) &= a, \\ e_1 &= f(x) - f(y) \end{aligned} \longrightarrow \begin{aligned} f(e_1) &= a \\ e_1 &= e_2 - e_3 \\ e_2 &= f(x) \\ e_3 &= f(y) \end{aligned}$$

$$f(0) > a + 2 \longrightarrow \begin{aligned} f(e_4) &= a + 2, \\ e_4 &= 0 \end{aligned} \longrightarrow \begin{aligned} f(e_4) &= e_5 \\ e_4 &= 0 \\ e_5 &> a + 2 \end{aligned}$$

Step 2: Exchange entailed **interface equalities** (over shared constants $e_1, e_2, e_3, e_4, e_5, a$).

$$L_1 = \{f(e_1) = a,\ f(x) = e_2,\ f(y) = e_3,\ f(e_4) = e_5,\ x = y,\ e_1 = e_4\}$$

$$L_2 = \{e_2 - e_3 = e_1,\ e_4 = 0,\ e_5 > a + 2,\ e_2 = e_3,\ a = e_5\}$$

$$\dfrac{f(e_1) = a \quad f(e_4) = e_5 \quad \dfrac{e_4 = 0 \quad e_2 - e_3 = e_1 \quad \dfrac{f(x) = e_2 \quad f(y) = e_3 \quad x = y}{e_2 = e_3}\ \mathcal{T}_{\text{UF}}}{e_1 = e_4}\ \mathcal{T}_{\text{LRA}}}{a = e_5}\ \mathcal{T}_{\text{UF}}$$

Step 3: Check for satisfiability.

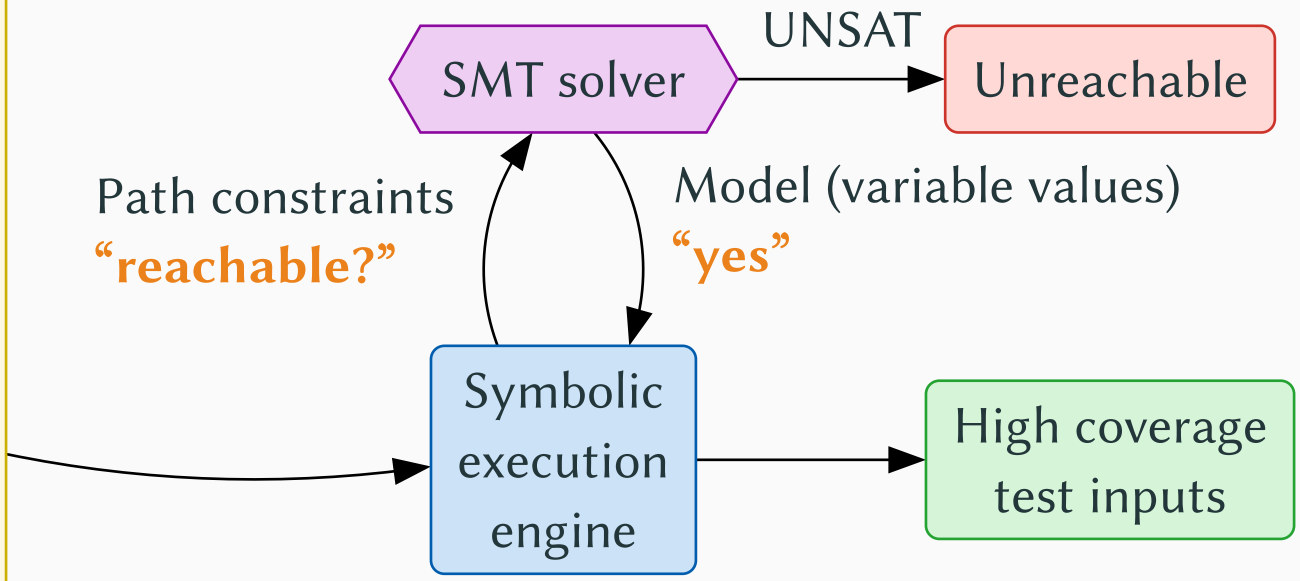
- $L_1 \not\models_{\text{UF}} \bot$
- $L_2 \models_{\text{LRA}} \bot$
- Thus, the whole formula is **unsatisfiable**.

$$\dfrac{e_5 > a + 2 \quad a = e_5}{\bot}\ \mathcal{T}_{\text{LRA}}$$

4. Applications of SMT

```
void func(int x, int y) {
  int z = 2 * y;
  if (z == x) {
    if (x > y + 10) {
      assert(false); // !
    }
  }
}

int main() {
  int x = sym_input();
  int y = sym_input();
  func(x, y);
  return 0;
}
```



Code	Symbolic state
`def func(x: int, y: int):`	
`    z = 2 * y`	$x \mapsto x_0$ $y \mapsto y_0$ $z \mapsto 2 \cdot y_0$
`    if x == z:`	PC: $(x_0 = 2 \cdot y_0)$
`        if x > y + 10:`	PC: $(x_0 = 2 \cdot y_0) \land (x_0 > y_0 + 10)$
`            raise RuntimeError("???")`	Reachable with $x_0 = 40, y_0 = 20$
`def main():`	
`    x = sym_input()`	$x \mapsto x_0$
`    y = sym_input()`	$x \mapsto x_0$ $y \mapsto y_0$
`    func(x, y)`	PC: $\top$

5. Conclusion

5. Conclusion

- SMT **extends** the capabilities of SAT by incorporating **rich theories**.
- SMT solvers are **precise** which makes them a critical tool for verification and analysis.
- As SMT **evolves**, its role in modern computing is becoming more prominent.

Thanks.

kchukharev@itmo.ru

Lipen