CS498: Algorithmic Engineering

Lecture 19: SMT = SAT + Theories

Elfarouk Harb

University of Illinois Urbana-Champaign

Week 11 – 03/26/2026

Outline

Where We Left Off

Lecture 17: we used Z3 to solve puzzles with **Bool** variables.

- Declared `Bool` variables, wrote clauses, hit `s.check()`.

Where We Left Off

Lecture 17: we used Z3 to solve puzzles with **Bool** variables.

- Declared `Bool` variables, wrote clauses, hit `s.check()`.
- Party puzzle, Sudoku, *n*-queens. All solved in milliseconds.

Where We Left Off

Lecture 17: we used Z3 to solve puzzles with **Bool** variables.

- Declared `Bool` variables, wrote clauses, hit `s.check()`.
- Party puzzle, Sudoku, *n*-queens. All solved in milliseconds.

Lecture 18: we opened the hood for **SAT**.

- DPLL: systematic search with unit propagation and pure literal elimination.

Where We Left Off

Lecture 17: we used Z3 to solve puzzles with **Bool** variables.

- Declared `Bool` variables, wrote clauses, hit `s.check()`.
- Party puzzle, Sudoku, *n*-queens. All solved in milliseconds.

Lecture 18: we opened the hood for **SAT**.

- DPLL: systematic search with unit propagation and pure literal elimination.
- CDCL: learn conflict clauses to avoid repeating mistakes.

Where We Left Off

Lecture 17: we used Z3 to solve puzzles with **Bool** variables.

- Declared `Bool` variables, wrote clauses, hit `s.check()`.
- Party puzzle, Sudoku, *n*-queens. All solved in milliseconds.

Lecture 18: we opened the hood for **SAT**.

- DPLL: systematic search with unit propagation and pure literal elimination.
- CDCL: learn conflict clauses to avoid repeating mistakes.
- Modern SAT solvers handle millions of variables, billions of clauses.

Where We Left Off

Lecture 17: we used Z3 to solve puzzles with **Bool** variables.

- Declared `Bool` variables, wrote clauses, hit `s.check()`.
- Party puzzle, Sudoku, *n*-queens. All solved in milliseconds.

Lecture 18: we opened the hood for **SAT**.

- DPLL: systematic search with unit propagation and pure literal elimination.
- CDCL: learn conflict clauses to avoid repeating mistakes.
- Modern SAT solvers handle millions of variables, billions of clauses.

So we understand SAT. The question is: what happens when the problem is not purely boolean?

SAT Only Knows True and False

So far, every variable has been boolean: $x_1 \in \{\text{True}, \text{False}\}$.

SAT Only Knows True and False

So far, every variable has been boolean: $x_1 \in \{\text{True}, \text{False}\}$.

But: what if we want to write things like this?

- `x = Int('x')` (integer variable, not boolean!)

SAT Only Knows True and False

So far, every variable has been boolean: $x_1 \in \{\text{True}, \text{False}\}$.

But: what if we want to write things like this?

- `x = Int('x')` (integer variable, not boolean!)
- `s.add(x + 2*y == 10)` (arithmetic, not logic!)

SAT Only Knows True and False

So far, every variable has been boolean: $x_1 \in \{\text{True}, \text{False}\}$.

But: what if we want to write things like this?

- `x = Int('x')` (integer variable, not boolean!)
- `s.add(x + 2*y == 10)` (arithmetic, not logic!)
- `s.add(Distinct(a, b, c))` (all-different constraint)

SAT Only Knows True and False

So far, every variable has been boolean: $x_1 \in \{\text{True}, \text{False}\}$.

But: what if we want to write things like this?

- `x = Int('x')` (integer variable, not boolean!)
- `s.add(x + 2*y == 10)` (arithmetic, not logic!)
- `s.add(Distinct(a, b, c))` (all-different constraint)

None of that is boolean satisfiability. A SAT solver has no idea what "$+$" or "$<$" means.

SAT Only Knows True and False

So far, every variable has been boolean: $x_1 \in \{\text{True}, \text{False}\}$.

But: what if we want to write things like this?

- `x = Int('x')` (integer variable, not boolean!)
- `s.add(x + 2*y == 10)` (arithmetic, not logic!)
- `s.add(Distinct(a, b, c))` (all-different constraint)

None of that is boolean satisfiability. A SAT solver has no idea what "$+$" or "$<$" means.

The gap

CDCL speaks boolean. Our problems speak integers, arrays, functions. How to bridge?

A Motivating Example

Suppose we want to solve:

$$x + 2y = 9, \quad x > 0, \quad y > 0, \quad x \leq y, \quad f(x) = f(y) \quad \forall f, \quad x \in \mathbb{Z} \quad y \in \mathbb{Z}.$$

A Motivating Example

Suppose we want to solve:

$$x + 2y = 9, \quad x > 0, \quad y > 0, \quad x \leq y, \quad f(x) = f(y) \quad \forall f, \quad x \in \mathbb{Z} \quad y \in \mathbb{Z}.$$

Integers, addition, functions, comparison. Nothing boolean.

A Motivating Example

Suppose we want to solve:

$$x + 2y = 9, \quad x > 0, \quad y > 0, \quad x \leq y, \quad f(x) = f(y) \quad \forall f, \quad x \in \mathbb{Z} \quad y \in \mathbb{Z}.$$

Integers, addition, functions, comparison. Nothing boolean.

The answer: SMT (Satisfiability Modulo Theories).

A Motivating Example

Suppose we want to solve:

$$x + 2y = 9, \quad x > 0, \quad y > 0, \quad x \leq y, \quad f(x) = f(y) \quad \forall f, \quad x \in \mathbb{Z} \quad y \in \mathbb{Z}.$$

Integers, addition, functions, comparison. Nothing boolean.

The answer: SMT (Satisfiability Modulo Theories).

Keep the SAT engine for boolean reasoning, plug in **specialized solvers** for arithmetic, arrays, functions, etc. Let them collaborate.

Starting from the Problem

What does it mean for a solver to “understand” integers?

Starting from the Problem

What does it mean for a solver to “understand” integers?

Consider:

$$x + y \leq 5, \quad x \geq 2, \quad y \geq 4.$$

Starting from the Problem

What does it mean for a solver to “understand” integers?

Consider:

$$x + y \leq 5, \quad x \geq 2, \quad y \geq 4.$$

- $x \geq 2$ and $y \geq 4 \Rightarrow x + y \geq 6$.

Starting from the Problem

What does it mean for a solver to “understand” integers?

Consider:

$$x + y \leq 5, \quad x \geq 2, \quad y \geq 4.$$

- $x \geq 2$ and $y \geq 4 \Rightarrow x + y \geq 6$.
- But $x + y \leq 5$. Contradiction. **UNSAT.**

Starting from the Problem

What does it mean for a solver to “understand” integers?

Consider:

$$x + y \leq 5, \quad x \geq 2, \quad y \geq 4.$$

- $x \geq 2$ and $y \geq 4 \Rightarrow x + y \geq 6$.
- But $x + y \leq 5$. Contradiction. **UNSAT.**

A SAT solver cannot do this: it used the *meaning* of $+$, $\leq$, $\geq$ over integers.

Starting from the Problem

What does it mean for a solver to “understand” integers?

Consider:

$$x + y \leq 5, \quad x \geq 2, \quad y \geq 4.$$

- $x \geq 2$ and $y \geq 4 \Rightarrow x + y \geq 6$.
- But $x + y \leq 5$. Contradiction. **UNSAT.**

A SAT solver cannot do this: it used the *meaning* of $+, \leq, \geq$ over integers.

A theory solver = specialized engine that answers “is this conjunction of constraints feasible?” for a specific kind of constraint.

What Is a Theory? (Formally)

A **theory** consists of:

1. A set of **symbols**: variables, constants, operations, relations.

What Is a Theory? (Formally)

A **theory** consists of:

1. A set of **symbols**: variables, constants, operations, relations.
2. A set of **axioms**: rules these symbols must obey.

What Is a Theory? (Formally)

A **theory** consists of:

1. A set of **symbols**: variables, constants, operations, relations.
2. A set of **axioms**: rules these symbols must obey.

Theory Solver

Given a conjunction $c_1 \wedge c_2 \wedge \cdots \wedge c_k$ of T-atoms, is there a satisfying assignment under the rules of T?

What Is a Theory? (Formally)

A **theory** consists of:

1. A set of **symbols**: variables, constants, operations, relations.
2. A set of **axioms**: rules these symbols must obey.

Theory Solver

Given a conjunction $c_1 \wedge c_2 \wedge \cdots \wedge c_k$ of T-atoms, is there a satisfying assignment under the rules of T?

A specialized feasibility checker for one kind of constraint.

Theory 1: Linear Integer Arithmetic (LIA)

Symbols:

- Integer variables: $x, y, z, \ldots$

Theory 1: Linear Integer Arithmetic (LIA)

Symbols:

- Integer variables: $x, y, z, \ldots$
- Operations: $+$, $-$, $\times$ (by constants only, e.g., $3x$, not $x \cdot y$).

Theory 1: Linear Integer Arithmetic (LIA)

Symbols:

- Integer variables: $x, y, z, \ldots$
- Operations: $+$, $-$, $\times$ (by constants only, e.g., $3x$, not $x \cdot y$).
- Relations: $=$, $<$, $\leq$, $>$, $\geq$.

Theory 1: Linear Integer Arithmetic (LIA)

Symbols:

- Integer variables: $x, y, z, \ldots$
- Operations: $+$, $-$, $\times$ (by constants only, e.g., $3x$, not $x \cdot y$).
- Relations: $=$, $<$, $\leq$, $>$, $\geq$.
- Integer constants: $0, 1, 2, \ldots$

Theory 1: Linear Integer Arithmetic (LIA)

Symbols:

- Integer variables: $x, y, z, \ldots$
- Operations: $+$, $-$, $\times$ (by constants only, e.g., $3x$, not $x \cdot y$).
- Relations: $=, <, \leq, >, \geq$.
- Integer constants: $0, 1, 2, \ldots$

A LIA theory solver takes a conjunction like:

$$2x + 3y \leq 12, \quad x \geq 1, \quad y \geq 1, \quad x + y \geq 5, \quad x \in \mathbb{Z} \quad y \in \mathbb{Z}$$

and decides: is it feasible?

Theory 1: Linear Integer Arithmetic (LIA)

Symbols:

- Integer variables: $x, y, z, \ldots$
- Operations: $+$, $-$, $\times$ (by constants only, e.g., $3x$, not $x \cdot y$).
- Relations: $=, <, \leq, >, \geq$.
- Integer constants: $0, 1, 2, \ldots$

A LIA theory solver takes a conjunction like:

$$2x + 3y \leq 12, \quad x \geq 1, \quad y \geq 1, \quad x + y \geq 5, \quad x \in \mathbb{Z} \quad y \in \mathbb{Z}$$

and decides: is it feasible?

Sound familiar? Same idea as integer programming from Part I. The theory solver is a mini-Gurobi (branch-and-bound + simplex) for this fragment.

Theory 1: Linear Integer Arithmetic (LIA)

Symbols:

- Integer variables: $x, y, z, \ldots$
- Operations: $+$, $-$, $\times$ (by constants only, e.g., $3x$, not $x \cdot y$).
- Relations: $=$, $<$, $\leq$, $>$, $\geq$.
- Integer constants: $0, 1, 2, \ldots$

A LIA theory solver takes a conjunction like:

$$2x + 3y \leq 12, \quad x \geq 1, \quad y \geq 1, \quad x + y \geq 5, \quad x \in \mathbb{Z} \quad y \in \mathbb{Z}$$

and decides: is it feasible?

Sound familiar? Same idea as integer programming from Part I. The theory solver is a mini-Gurobi (branch-and-bound + simplex) for this fragment.

Also: **LRA** (Linear Real Arithmetic) — same symbols, variables over $\mathbb{R}$. Easier (LP instead of ILP).

Theory 2: Equality with Uninterpreted Functions (EUF)

Symbols:

- Variables: $a, b, c, \ldots$

Theory 2: Equality with Uninterpreted Functions (EUF)

Symbols:

- Variables: $a, b, c, \ldots$
- Function symbols: $f, g, h, \ldots$ (we do not know what they compute).

Theory 2: Equality with Uninterpreted Functions (EUF)

Symbols:

- Variables: $a, b, c, \ldots$
- Function symbols: $f, g, h, \ldots$ (we do not know what they compute).
- Relations: $=$ and $\neq$ only.

Theory 2: Equality with Uninterpreted Functions (EUF)

Symbols:

- Variables: $a, b, c, \ldots$
- Function symbols: $f, g, h, \ldots$ (we do not know what they compute).
- Relations: $=$ and $\neq$ only.

The key axiom (congruence):

$$\boxed{a = b \;\Rightarrow\; f(a) = f(b)}$$

Theory 2: Equality with Uninterpreted Functions (EUF)

Symbols:

- Variables: $a, b, c, \ldots$
- Function symbols: $f, g, h, \ldots$ (we do not know what they compute).
- Relations: $=$ and $\neq$ only.

The key axiom (congruence):

$$\boxed{a = b \;\Rightarrow\; f(a) = f(b)}$$

We don't know what f computes. We just know: same input $\Rightarrow$ same output.

Theory 2: Equality with Uninterpreted Functions (EUF)

Symbols:

- Variables: $a, b, c, \ldots$
- Function symbols: $f, g, h, \ldots$ (we do not know what they compute).
- Relations: $=$ and $\neq$ only.

The key axiom (congruence):

$$\boxed{a = b \;\Rightarrow\; f(a) = f(b)}$$

We don't know what f computes. We just know: same input $\Rightarrow$ same output.

- $\{a = b,\ f(a) \neq f(b)\}$: **UNSAT**. Congruence forces $f(a) = f(b)$.

Theory 2: Equality with Uninterpreted Functions (EUF)

Symbols:

- Variables: $a, b, c, \ldots$
- Function symbols: $f, g, h, \ldots$ (we do not know what they compute).
- Relations: $=$ and $\neq$ only.

The key axiom (congruence):

$$\boxed{a = b \;\Rightarrow\; f(a) = f(b)}$$

We don't know what f computes. We just know: same input $\Rightarrow$ same output.

- $\{a = b,\ f(a) \neq f(b)\}$: **UNSAT**. Congruence forces $f(a) = f(b)$.
- $\{a \neq b,\ f(a) = f(b)\}$: **SAT**. f could be constant.

Theory 2: Equality with Uninterpreted Functions (EUF)

Symbols:

- Variables: $a, b, c, \ldots$
- Function symbols: $f, g, h, \ldots$ (we do not know what they compute).
- Relations: $=$ and $\neq$ only.

The key axiom (congruence):

$$\boxed{a = b \;\Rightarrow\; f(a) = f(b)}$$

We don't know what f computes. We just know: same input $\Rightarrow$ same output.

- $\{a = b,\ f(a) \neq f(b)\}$: **UNSAT**. Congruence forces $f(a) = f(b)$.
- $\{a \neq b,\ f(a) = f(b)\}$: **SAT**. f could be constant.

Implementation: union-find (disjoint set) data structure. Learn $a = b$? Union their sets. Check equality? Compare representatives.

Why Uninterpreted Functions?

Use case: software verification.

Why Uninterpreted Functions?

Use case: software verification.

Two code paths call the same function:

Path A:

```
z = compute(x)
```

Why Uninterpreted Functions?

Use case: software verification.

Two code paths call the same function:

Path A:

```
z = compute(x)
```

Path B:

```
w = compute(x)
```

Why Uninterpreted Functions?

Use case: software verification.

Two code paths call the same function:

Path A:

```
z = compute(x)
```

Path B:

```
w = compute(x)
```

Want to verify $z = w$ without analyzing `compute`'s internals.

Why Uninterpreted Functions?

Use case: software verification.

Two code paths call the same function:

Path A:

```
z = compute(x)
```

Path B:

```
w = compute(x)
```

Want to verify $z = w$ without analyzing `compute`'s internals.

Model `compute` as uninterpreted f. Then $z = f(x)$, $w = f(x)$, so $z = w$ by congruence.

Why Uninterpreted Functions?

Use case: software verification.

Two code paths call the same function:

Path A:

```
z = compute(x)
```

Path B:

```
w = compute(x)
```

Want to verify $z = w$ without analyzing `compute`'s internals.

Model `compute` as uninterpreted f. Then $z = f(x)$, $w = f(x)$, so $z = w$ by congruence.

Key: reason about *relationships between calls* without knowing what functions do.

The Running Example

We will trace one example through three approaches.

The Running Example

We will trace one example through three approaches.

Running example:

$$\boxed{(x > 0 \ \text{OR} \ y > 0) \ \text{AND} \ (x + y < 3) \ \text{AND} \ (x = y)}$$

The Running Example

We will trace one example through three approaches.

Running example:

$$\boxed{(x > 0 \ \text{OR} \ y > 0) \ \text{AND} \ (x + y < 3) \ \text{AND} \ (x = y)}$$

Boolean structure (OR, AND) mixed with **theory atoms** ($x > 0$, $x + y < 3$, $x = y$).

The Running Example

We will trace one example through three approaches.

Running example:

$$\boxed{(x > 0 \ \text{OR} \ y > 0) \ \text{AND} \ (x + y < 3) \ \text{AND} \ (x = y)}$$

Boolean structure (OR, AND) mixed with **theory atoms** ($x > 0$, $x + y < 3$, $x = y$).

Note the OR! This is *not* a pure ILP. An ILP is a conjunction of linear constraints; no disjunctions. Here we need **boolean reasoning on top of arithmetic**.

The Running Example

We will trace one example through three approaches.

Running example:

$$\boxed{(x > 0 \ \text{OR} \ y > 0) \ \text{AND} \ (x + y < 3) \ \text{AND} \ (x = y)}$$

Boolean structure (OR, AND) mixed with **theory atoms** ($x > 0$, $x + y < 3$, $x = y$).

Note the OR! This is *not* a pure ILP. An ILP is a conjunction of linear constraints; no disjunctions. Here we need **boolean reasoning on top of arithmetic**.

Three approaches:

1. **Eager** (bit-blasting): translate everything to pure SAT.

The Running Example

We will trace one example through three approaches.

Running example:

$$\boxed{(x > 0 \ \text{OR} \ y > 0) \ \text{AND} \ (x + y < 3) \ \text{AND} \ (x = y)}$$

Boolean structure (OR, AND) mixed with **theory atoms** ($x > 0$, $x + y < 3$, $x = y$).

Note the OR! This is *not* a pure ILP. An ILP is a conjunction of linear constraints; no disjunctions. Here we need **boolean reasoning on top of arithmetic**.

Three approaches:

1. **Eager** (bit-blasting): translate everything to pure SAT.
2. **Lazy**: SAT solver and theory solver take turns.

The Running Example

We will trace one example through three approaches.

Running example:

$$\boxed{(x > 0 \ \text{OR} \ y > 0) \ \text{AND} \ (x + y < 3) \ \text{AND} \ (x = y)}$$

Boolean structure (OR, AND) mixed with **theory atoms** ($x > 0$, $x + y < 3$, $x = y$).

Note the OR! This is *not* a pure ILP. An ILP is a conjunction of linear constraints; no disjunctions. Here we need **boolean reasoning on top of arithmetic**.

Three approaches:

1. **Eager** (bit-blasting): translate everything to pure SAT.
2. **Lazy**: SAT solver and theory solver take turns.
3. **DPLL(T)**: tight collaboration. What Z3 does*.

Approach 1: Eager (Bit-Blasting)

Idea: forget about theory solvers entirely. Translate EVERYTHING into pure boolean SAT.

Approach 1: Eager (Bit-Blasting)

Idea: forget about theory solvers entirely. Translate EVERYTHING into pure boolean SAT.

Step 1: represent each integer as a fixed-width bitvector. Say, 4 bits (signed).

$$x \leftrightarrow (x_3, x_2, x_1, x_0), \qquad y \leftrightarrow (y_3, y_2, y_1, y_0)$$

Approach 1: Eager (Bit-Blasting)

Idea: forget about theory solvers entirely. Translate EVERYTHING into pure boolean SAT.

Step 1: represent each integer as a fixed-width bitvector. Say, 4 bits (signed).

$$x \leftrightarrow (x_3, x_2, x_1, x_0), \qquad y \leftrightarrow (y_3, y_2, y_1, y_0)$$

Step 2: encode arithmetic as boolean circuits.

- $x + y$: ripple-carry adder (chain of full adders).

Approach 1: Eager (Bit-Blasting)

Idea: forget about theory solvers entirely. Translate EVERYTHING into pure boolean SAT.

Step 1: represent each integer as a fixed-width bitvector. Say, 4 bits (signed).

$$x \leftrightarrow (x_3, x_2, x_1, x_0), \qquad y \leftrightarrow (y_3, y_2, y_1, y_0)$$

Step 2: encode arithmetic as boolean circuits.

- $x + y$: ripple-carry adder (chain of full adders).
- $x > 0$: check sign bit and whether at least one bit is 1.

Approach 1: Eager (Bit-Blasting)

Idea: forget about theory solvers entirely. Translate EVERYTHING into pure boolean SAT.

Step 1: represent each integer as a fixed-width bitvector. Say, 4 bits (signed).

$$x \leftrightarrow (x_3, x_2, x_1, x_0), \qquad y \leftrightarrow (y_3, y_2, y_1, y_0)$$

Step 2: encode arithmetic as boolean circuits.

- $x + y$: ripple-carry adder (chain of full adders).
- $x > 0$: check sign bit and whether at least one bit is 1.
- $x = y$: bitwise equality, $\bigwedge_{i=0}^{3}(x_i \leftrightarrow y_i)$.

Approach 1: Eager (Bit-Blasting)

Idea: forget about theory solvers entirely. Translate EVERYTHING into pure boolean SAT.

Step 1: represent each integer as a fixed-width bitvector. Say, 4 bits (signed).

$$x \leftrightarrow (x_3, x_2, x_1, x_0), \qquad y \leftrightarrow (y_3, y_2, y_1, y_0)$$

Step 2: encode arithmetic as boolean circuits.

- $x + y$: ripple-carry adder (chain of full adders).
- $x > 0$: check sign bit and whether at least one bit is 1.
- $x = y$: bitwise equality, $\bigwedge_{i=0}^{3}(x_i \leftrightarrow y_i)$.

Step 3: the entire formula becomes a CNF over the bit variables. Hand it to a SAT solver.

Approach 1: Eager (Bit-Blasting)

Idea: forget about theory solvers entirely. Translate EVERYTHING into pure boolean SAT.

Step 1: represent each integer as a fixed-width bitvector. Say, 4 bits (signed).

$$x \leftrightarrow (x_3, x_2, x_1, x_0), \qquad y \leftrightarrow (y_3, y_2, y_1, y_0)$$

Step 2: encode arithmetic as boolean circuits.

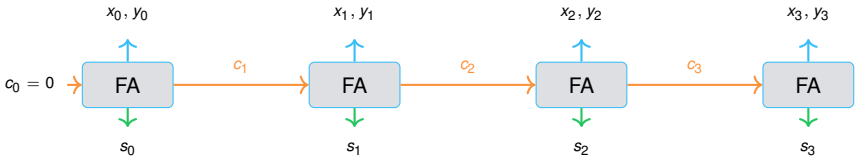
- $x + y$: ripple-carry adder (chain of full adders).
- $x > 0$: check sign bit and whether at least one bit is 1.
- $x = y$: bitwise equality, $\bigwedge_{i=0}^{3}(x_i \leftrightarrow y_i)$.

Step 3: the entire formula becomes a CNF over the bit variables. Hand it to a SAT solver.

Result: a standard SAT problem. CDCL can solve it.

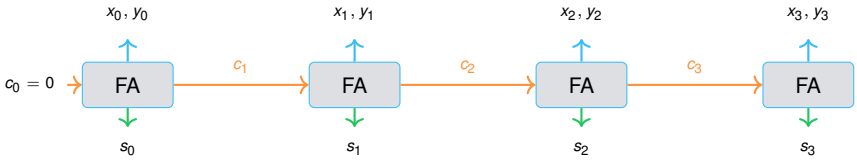
Bit-Blasting: What the Adder Looks Like

A **full adder** for one bit position:



Bit-Blasting: What the Adder Looks Like

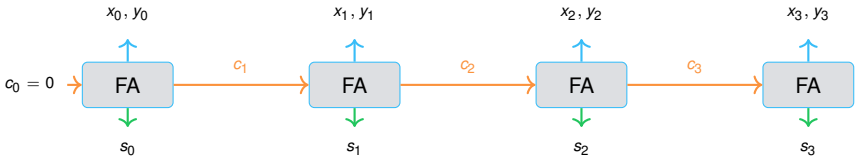
A **full adder** for one bit position:



Each FA: $s_i = x_i \oplus y_i \oplus c_i, \quad c_{i+1} = \text{majority}(x_i, y_i, c_i)$.

Bit-Blasting: What the Adder Looks Like

A **full adder** for one bit position:



Each FA: $s_i = x_i \oplus y_i \oplus c_i, \quad c_{i+1} = \text{majority}(x_i, y_i, c_i)$.

4 bits: $\sim$20+ boolean variables just for addition. Now imagine 32- or 64-bit integers.

Bit-Blasting: Pros and Cons

Pros:

- Conceptually simple.

Bit-Blasting: Pros and Cons

Pros:

- Conceptually simple.
- Works for any finite-domain theory.

Bit-Blasting: Pros and Cons

Pros:

- Conceptually simple.
- Works for any finite-domain theory.
- Leverages decades of SAT solver engineering.

Bit-Blasting: Pros and Cons

Pros:

- Conceptually simple.
- Works for any finite-domain theory.
- Leverages decades of SAT solver engineering.

Cons:

- Formula blows up (32 booleans per 32-bit int).

Bit-Blasting: Pros and Cons

Pros:

- Conceptually simple.
- Works for any finite-domain theory.
- Leverages decades of SAT solver engineering.

Cons:

- Formula blows up (32 booleans per 32-bit int).
- Loses arithmetic structure.

Bit-Blasting: Pros and Cons

Pros:

- Conceptually simple.
- Works for any finite-domain theory.
- Leverages decades of SAT solver engineering.

Cons:

- Formula blows up (32 booleans per 32-bit int).
- Loses arithmetic structure.
- Cannot handle unbounded integers.

Bit-Blasting: Pros and Cons

Pros:

- Conceptually simple.
- Works for any finite-domain theory.
- Leverages decades of SAT solver engineering.

Cons:

- Formula blows up (32 booleans per 32-bit int).
- Loses arithmetic structure.
- Cannot handle unbounded integers.

But:

Bit-blasting *is* the method of choice for **bitvector** theories (fixed-width machine integers). For general integer arithmetic, we need something smarter.

Approach 2: The Lazy Approach

Idea: treat theory atoms as opaque boolean variables. Solve the boolean skeleton, then check if the theory is happy.

Approach 2: The Lazy Approach

Idea: treat theory atoms as opaque boolean variables. Solve the boolean skeleton, then check if the theory is happy.

Recall:

$$(x > 0 \text{ OR } y > 0) \text{ AND } (x + y < 3) \text{ AND } (x = y)$$

Approach 2: The Lazy Approach

Idea: treat theory atoms as opaque boolean variables. Solve the boolean skeleton, then check if the theory is happy.

Recall:

$$(x > 0 \text{ OR } y > 0) \text{ AND } (x + y < 3) \text{ AND } (x = y)$$

Four phases:

1. Boolean abstraction.

Approach 2: The Lazy Approach

Idea: treat theory atoms as opaque boolean variables. Solve the boolean skeleton, then check if the theory is happy.

Recall:

$$(x > 0 \text{ OR } y > 0) \text{ AND } (x + y < 3) \text{ AND } (x = y)$$

Four phases:

1. Boolean abstraction.
2. SAT solving on the skeleton.

Approach 2: The Lazy Approach

Idea: treat theory atoms as opaque boolean variables. Solve the boolean skeleton, then check if the theory is happy.

Recall:

$$(x > 0 \text{ OR } y > 0) \text{ AND } (x + y < 3) \text{ AND } (x = y)$$

Four phases:

1. Boolean abstraction.
2. SAT solving on the skeleton.
3. Theory checking.

Approach 2: The Lazy Approach

Idea: treat theory atoms as opaque boolean variables. Solve the boolean skeleton, then check if the theory is happy.

Recall:

$$(x > 0 \text{ OR } y > 0) \text{ AND } (x + y < 3) \text{ AND } (x = y)$$

Four phases:

1. Boolean abstraction.
2. SAT solving on the skeleton.
3. Theory checking.
4. Refinement (if theory says no).

Lazy Step 1: Boolean Abstraction

Replace each **theory atom** with a fresh boolean variable:

Lazy Step 1: Boolean Abstraction

Replace each **theory atom** with a fresh boolean variable:

Boolean variable		**Theory atom**
b_1	$\leftrightarrow$	$x > 0$
b_2	$\leftrightarrow$	$y > 0$
b_3	$\leftrightarrow$	$x + y < 3$
b_4	$\leftrightarrow$	$x = y$

Lazy Step 1: Boolean Abstraction

Replace each **theory atom** with a fresh boolean variable:

Boolean variable		**Theory atom**
b_1	$\leftrightarrow$	$x > 0$
b_2	$\leftrightarrow$	$y > 0$
b_3	$\leftrightarrow$	$x + y < 3$
b_4	$\leftrightarrow$	$x = y$

The original formula becomes:

$$\boxed{(b_1 \vee b_2) \ \wedge \ b_3 \ \wedge \ b_4}$$

Lazy Step 1: Boolean Abstraction

Replace each **theory atom** with a fresh boolean variable:

Boolean variable		Theory atom
b_1	$\leftrightarrow$	$x > 0$
b_2	$\leftrightarrow$	$y > 0$
b_3	$\leftrightarrow$	$x + y < 3$
b_4	$\leftrightarrow$	$x = y$

The original formula becomes:

$$\boxed{(b_1 \vee b_2) \ \wedge \ b_3 \ \wedge \ b_4}$$

Pure SAT. The solver has no idea what b_1 "means": just booleans and clauses.

Lazy Step 2: SAT Solver Finds a Boolean Assignment

The SAT solver works on $(b_1 \vee b_2) \wedge b_3 \wedge b_4$.

Lazy Step 2: SAT Solver Finds a Boolean Assignment

The SAT solver works on $(b_1 \vee b_2) \wedge b_3 \wedge b_4$.

It must set $b_3 = \mathrm{T}$ and $b_4 = \mathrm{T}$ (unit clauses).

Lazy Step 2: SAT Solver Finds a Boolean Assignment

The SAT solver works on $(b_1 \vee b_2) \wedge b_3 \wedge b_4$.

It must set $b_3 = \mathrm{T}$ and $b_4 = \mathrm{T}$ (unit clauses).
For $b_1 \vee b_2$, it picks (say) $b_1 = \mathrm{T}$, $b_2 = \mathrm{T}$.

Lazy Step 2: SAT Solver Finds a Boolean Assignment

The SAT solver works on $(b_1 \vee b_2) \wedge b_3 \wedge b_4$.

It must set $b_3 = \mathsf{T}$ and $b_4 = \mathsf{T}$ (unit clauses).
For $b_1 \vee b_2$, it picks (say) $b_1 = \mathsf{T}$, $b_2 = \mathsf{T}$.

Boolean assignment: $b_1 = \mathsf{T},\ b_2 = \mathsf{T},\ b_3 = \mathsf{T},\ b_4 = \mathsf{T}$.

Lazy Step 2: SAT Solver Finds a Boolean Assignment

The SAT solver works on $(b_1 \vee b_2) \wedge b_3 \wedge b_4$.

It must set $b_3 = \mathrm{T}$ and $b_4 = \mathrm{T}$ (unit clauses).
For $b_1 \vee b_2$, it picks (say) $b_1 = \mathrm{T}$, $b_2 = \mathrm{T}$.

Boolean assignment: $b_1 = \mathrm{T},\ b_2 = \mathrm{T},\ b_3 = \mathrm{T},\ b_4 = \mathrm{T}$.

Decode back to theory atoms:

$$x > 0, \quad y > 0, \quad x + y < 3, \quad x = y.$$

Lazy Step 2: SAT Solver Finds a Boolean Assignment

The SAT solver works on $(b_1 \vee b_2) \wedge b_3 \wedge b_4$.

It must set $b_3 = \mathsf{T}$ and $b_4 = \mathsf{T}$ (unit clauses).
For $b_1 \vee b_2$, it picks (say) $b_1 = \mathsf{T}$, $b_2 = \mathsf{T}$.

Boolean assignment: $b_1 = \mathsf{T},\ b_2 = \mathsf{T},\ b_3 = \mathsf{T},\ b_4 = \mathsf{T}$.

Decode back to theory atoms:

$$x > 0, \quad y > 0, \quad x + y < 3, \quad x = y.$$

Does this conjunction have an integer solution? Hand it to the **LIA theory solver**.

Lazy Step 3: Theory Solver Checks

LIA theory solver receives:

$$x > 0 \ \wedge \ y > 0 \ \wedge \ x + y < 3 \ \wedge \ x = y.$$

Lazy Step 3: Theory Solver Checks

LIA theory solver receives:

$$x > 0 \ \wedge \ y > 0 \ \wedge \ x + y < 3 \ \wedge \ x = y.$$

$x = y, x > 0 \Rightarrow x \geq 1, y \geq 1 \Rightarrow x + y \geq 2.$

Lazy Step 3: Theory Solver Checks

LIA theory solver receives:

$$x > 0 \ \wedge \ y > 0 \ \wedge \ x + y < 3 \ \wedge \ x = y.$$

$x = y$, $x > 0 \Rightarrow x \geq 1$, $y \geq 1 \Rightarrow x + y \geq 2$.
But $x + y < 3$, so $x + y = 2$, giving $x = 1, y = 1$.

Lazy Step 3: Theory Solver Checks

LIA theory solver receives:

$$x > 0 \ \wedge \ y > 0 \ \wedge \ x + y < 3 \ \wedge \ x = y.$$

$x = y$, $x > 0 \Rightarrow x \geq 1$, $y \geq 1 \Rightarrow x + y \geq 2$.
But $x + y < 3$, so $x + y = 2$, giving $x = 1, y = 1$.

Theory solver: SAT. Solution: $x = 1, y = 1$. **Done!**

Lazy Step 3: Theory Solver Checks

LIA theory solver receives:

$$x > 0 \ \wedge \ y > 0 \ \wedge \ x + y < 3 \ \wedge \ x = y.$$

$x = y$, $x > 0 \Rightarrow x \geq 1$, $y \geq 1 \Rightarrow x + y \geq 2$.
But $x + y < 3$, so $x + y = 2$, giving $x = 1, y = 1$.

Theory solver: SAT. Solution: $x = 1, y = 1$. **Done!**

Key

SAT solver handled booleans (OR, AND). Theory solver handled arithmetic. Teamwork.

What If the Theory Solver Says UNSAT?

Suppose the SAT solver had instead found:

$$b_1 = \mathsf{F},\ b_2 = \mathsf{T},\ b_3 = \mathsf{T},\ b_4 = \mathsf{T}.$$

What If the Theory Solver Says UNSAT?

Suppose the SAT solver had instead found:

$$b_1 = \mathsf{F},\ b_2 = \mathsf{T},\ b_3 = \mathsf{T},\ b_4 = \mathsf{T}.$$

Decoded: $x \leq 0,\ y > 0,\ x + y < 3,\ x = y.$

What If the Theory Solver Says UNSAT?

Suppose the SAT solver had instead found:

$$b_1 = \mathsf{F},\ b_2 = \mathsf{T},\ b_3 = \mathsf{T},\ b_4 = \mathsf{T}.$$

Decoded: $x \leq 0,\ y > 0,\ x + y < 3,\ x = y$.

$x = y$ and $x \leq 0 \Rightarrow y \leq 0$. But $y > 0$. **Contradiction. UNSAT.**

What If the Theory Solver Says UNSAT?

Suppose the SAT solver had instead found:

$$b_1 = \mathsf{F},\ b_2 = \mathsf{T},\ b_3 = \mathsf{T},\ b_4 = \mathsf{T}.$$

Decoded: $x \leq 0,\ y > 0,\ x + y < 3,\ x = y$.

$x = y$ and $x \leq 0 \Rightarrow y \leq 0$. But $y > 0$. **Contradiction. UNSAT.**

This assignment was bad, but maybe another works.

What If the Theory Solver Says UNSAT?

Suppose the SAT solver had instead found:

$$b_1 = \mathsf{F},\ b_2 = \mathsf{T},\ b_3 = \mathsf{T},\ b_4 = \mathsf{T}.$$

Decoded: $x \leq 0,\ y > 0,\ x + y < 3,\ x = y$.

$x = y$ and $x \leq 0 \Rightarrow y \leq 0$. But $y > 0$. **Contradiction. UNSAT.**

This assignment was bad, but maybe another works.

Fix: generate a **blocking clause** ("theory lemma") to prevent this mistake.

Lazy Step 4: Blocking Clauses

$\{\neg b_1, b_2, b_3, b_4\}$ is infeasible. Block it:

$$\neg(\neg b_1 \wedge b_2 \wedge b_3 \wedge b_4) \quad = \quad b_1 \vee \neg b_2 \vee \neg b_3 \vee \neg b_4$$

Lazy Step 4: Blocking Clauses

$\{\neg b_1, b_2, b_3, b_4\}$ is infeasible. Block it:

$$\neg(\neg b_1 \wedge b_2 \wedge b_3 \wedge b_4) \quad = \quad b_1 \vee \neg b_2 \vee \neg b_3 \vee \neg b_4$$

This **theory lemma** gets added to the clause database:

Lazy Step 4: Blocking Clauses

$\{\neg b_1, b_2, b_3, b_4\}$ is infeasible. Block it:

$$\neg(\neg b_1 \wedge b_2 \wedge b_3 \wedge b_4) \quad = \quad b_1 \vee \neg b_2 \vee \neg b_3 \vee \neg b_4$$

This **theory lemma** gets added to the clause database:

$$(b_1 \vee b_2) \ \wedge \ b_3 \ \wedge \ b_4 \ \wedge \ \underbrace{(b_1 \vee \neg b_2 \vee \neg b_3 \vee \neg b_4)}_{\text{theory lemma}}$$

Lazy Step 4: Blocking Clauses

$\{\neg b_1, b_2, b_3, b_4\}$ is infeasible. Block it:

$$\neg(\neg b_1 \wedge b_2 \wedge b_3 \wedge b_4) \quad = \quad b_1 \vee \neg b_2 \vee \neg b_3 \vee \neg b_4$$

This **theory lemma** gets added to the clause database:

$$(b_1 \vee b_2) \ \wedge \ b_3 \ \wedge \ b_4 \ \wedge \ \underbrace{(b_1 \vee \neg b_2 \vee \neg b_3 \vee \neg b_4)}_{\text{theory lemma}}$$

$b_3 = \mathsf{T}$, $b_4 = \mathsf{T} \Rightarrow$ lemma simplifies to b_1. Now $b_1 = \mathsf{T}$.

Lazy Step 4: Blocking Clauses

$\{\neg b_1, b_2, b_3, b_4\}$ is infeasible. Block it:

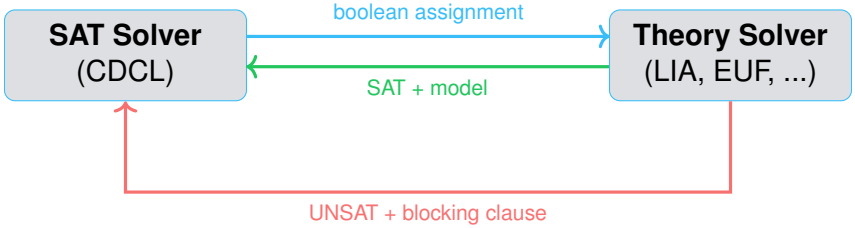
$$\neg(\neg b_1 \wedge b_2 \wedge b_3 \wedge b_4) \quad = \quad b_1 \vee \neg b_2 \vee \neg b_3 \vee \neg b_4$$

This **theory lemma** gets added to the clause database:

$$(b_1 \vee b_2) \ \wedge \ b_3 \ \wedge \ b_4 \ \wedge \ \underbrace{(b_1 \vee \neg b_2 \vee \neg b_3 \vee \neg b_4)}_{\text{theory lemma}}$$

$b_3 = \mathsf{T}$, $b_4 = \mathsf{T} \Rightarrow$ lemma simplifies to b_1. Now $b_1 = \mathsf{T}$.
SAT solver finds $b_1 = b_2 = b_3 = b_4 = \mathsf{T}$, theory confirms. Done.

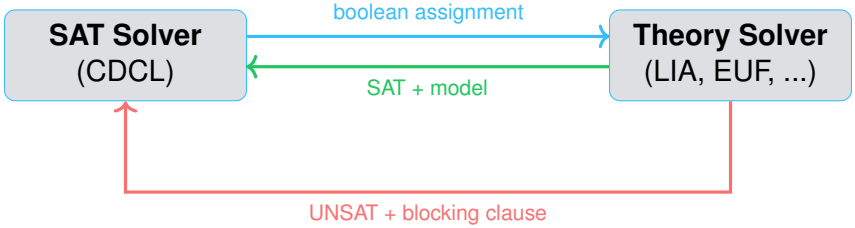
The Lazy Loop



The lazy loop:

1. SAT solver finds a boolean assignment (or reports UNSAT $\Rightarrow$ done).
2. Theory solver checks: feasible under the theory?
3. If yes: return SAT with the model.
4. If no: add a blocking clause, go back to step 1.

The Lazy Loop



The lazy loop:

1. SAT solver finds a boolean assignment (or reports UNSAT $\Rightarrow$ done).
2. Theory solver checks: feasible under the theory?
3. If yes: return SAT with the model.
4. If no: add a blocking clause, go back to step 1.

Terminates: finitely many boolean assignments; each clause eliminates ≥ 1.

The Problem with Lazy

Works, but can be slow.

The Problem with Lazy

Works, but can be slow.

Why? Theory solver only sees **complete** assignments. One at a time.

The Problem with Lazy

Works, but can be slow.

Why? Theory solver only sees **complete** assignments. One at a time.

100 theory atoms $\Rightarrow$ SAT solver may find thousands of boolean-SAT but theory-UNSAT assignments. One blocking clause each time.

The Problem with Lazy

Works, but can be slow.

Why? Theory solver only sees **complete** assignments. One at a time.

100 theory atoms $\Rightarrow$ SAT solver may find thousands of boolean-SAT but theory-UNSAT assignments. One blocking clause each time.

Can we catch theory conflicts *earlier*, before a full assignment?

The Problem with Lazy

Works, but can be slow.

Why? Theory solver only sees **complete** assignments. One at a time.

100 theory atoms $\Rightarrow$ SAT solver may find thousands of boolean-SAT but theory-UNSAT assignments. One blocking clause each time.

Can we catch theory conflicts *earlier*, before a full assignment?

Yes. $\Rightarrow$ DPLL(T).

Approach 3: DPLL(T)

Key idea: theory solver checks **partial** assignments during the search.

Approach 3: DPLL(T)

Key idea: theory solver checks **partial** assignments during the search.

Recall: CDCL builds assignments incrementally (decide, propagate, decide, propagate, ...).

Approach 3: DPLL(T)

Key idea: theory solver checks **partial** assignments during the search.

Recall: CDCL builds assignments incrementally (decide, propagate, decide, propagate, ...).

DPLL(T)/CDCL(T): every time the SAT solver sets a literal, it *notifies the theory solver*:

- "Already inconsistent" $\Rightarrow$ conflict, backtrack now.

Approach 3: DPLL(T)

Key idea: theory solver checks **partial** assignments during the search.

Recall: CDCL builds assignments incrementally (decide, propagate, decide, propagate, ...).

DPLL(T)/CDCL(T): every time the SAT solver sets a literal, it *notifies the theory solver*:

- "Already inconsistent" $\Rightarrow$ conflict, backtrack now.
- "This other literal must be true" $\Rightarrow$ theory propagation.

DPLL(T) on Our Running Example

$$(b_1 \vee b_2) \wedge b_3 \wedge b_4, \quad b_3 \leftrightarrow (x + y < 3), \quad b_4 \leftrightarrow (x = y).$$

DPLL(T) on Our Running Example

$(b_1 \vee b_2) \wedge b_3 \wedge b_4, \quad b_3 \leftrightarrow (x + y < 3), \quad b_4 \leftrightarrow (x = y).$

Step 1: unit prop $\Rightarrow b_3 = \mathrm{T}$, $b_4 = \mathrm{T}$. Notify theory solver.

DPLL(T) on Our Running Example

$(b_1 \vee b_2) \wedge b_3 \wedge b_4, \quad b_3 \leftrightarrow (x + y < 3), \quad b_4 \leftrightarrow (x = y).$

Step 1: unit prop $\Rightarrow b_3 = \mathrm{T}$, $b_4 = \mathrm{T}$. Notify theory solver.

Step 2: theory reasons:

- $x = y$ and $x + y < 3 \Rightarrow 2x < 3 \Rightarrow x \leq 1, y \leq 1, x = y$.

DPLL(T) on Our Running Example

$(b_1 \vee b_2) \wedge b_3 \wedge b_4, \quad b_3 \leftrightarrow (x + y < 3), \quad b_4 \leftrightarrow (x = y)$.

Step 1: unit prop $\Rightarrow b_3 = \mathrm{T}$, $b_4 = \mathrm{T}$. Notify theory solver.

Step 2: theory reasons:

- $x = y$ and $x + y < 3 \Rightarrow 2x < 3 \Rightarrow x \leq 1, y \leq 1, x = y$.

Step 3: SAT decides $b_1 = \mathrm{T}$ (for $b_1 \vee b_2$).

DPLL(T) on Our Running Example

$(b_1 \vee b_2) \wedge b_3 \wedge b_4, \quad b_3 \leftrightarrow (x + y < 3), \quad b_4 \leftrightarrow (x = y)$.

Step 1: unit prop $\Rightarrow b_3 = \mathrm{T}$, $b_4 = \mathrm{T}$. Notify theory solver.

Step 2: theory reasons:

- $x = y$ and $x + y < 3 \Rightarrow 2x < 3 \Rightarrow x \leq 1, y \leq 1, x = y$.

Step 3: SAT decides $b_1 = \mathrm{T}$ (for $b_1 \vee b_2$).
Theory: $x > 0$ and $x \leq 1 \Rightarrow x = 1$, $y = 1$. Propagate $b_2 = \mathrm{T}$.

DPLL(T) on Our Running Example

$(b_1 \vee b_2) \wedge b_3 \wedge b_4, \quad b_3 \leftrightarrow (x + y < 3), \quad b_4 \leftrightarrow (x = y).$

Step 1: unit prop $\Rightarrow b_3 = \mathrm{T}$, $b_4 = \mathrm{T}$. Notify theory solver.

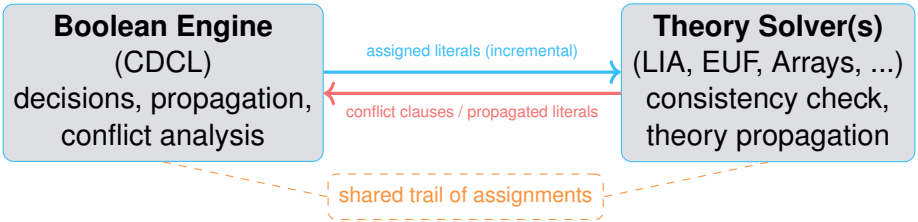
Step 2: theory reasons:

- $x = y$ and $x + y < 3 \Rightarrow 2x < 3 \Rightarrow x \leq 1$, $y \leq 1, x = y$.

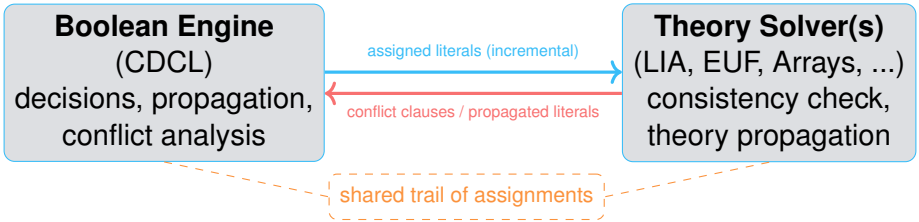
Step 3: SAT decides $b_1 = \mathrm{T}$ (for $b_1 \vee b_2$).
Theory: $x > 0$ and $x \leq 1 \Rightarrow x = 1$, $y = 1$. Propagate $b_2 = \mathrm{T}$.

Done in one pass. No round trips.

DPLL(T) Architecture



DPLL(T) Architecture



Tightly integrated. Information shared *during* search, not just at the end.

Three Approaches Compared

	Eager	**Lazy**	**DPLL(T)**
Formula size	Huge	Original	Original

Three Approaches Compared

	Eager	**Lazy**	**DPLL(T)**
Formula size	Huge	Original	Original
Theory consulted	Never	At the end	During search

Three Approaches Compared

	Eager	**Lazy**	**DPLL(T)**
Formula size	Huge	Original	Original
Theory consulted	Never	At the end	During search
Conflict detection	Late	Late	Early

Three Approaches Compared

	Eager	**Lazy**	**DPLL(T)**
Formula size	Huge	Original	Original
Theory consulted	Never	At the end	During search
Conflict detection	Late	Late	Early
Theory propagation	No	No	Yes

Three Approaches Compared

	Eager	**Lazy**	**DPLL(T)**
Formula size	Huge	Original	Original
Theory consulted	Never	At the end	During search
Conflict detection	Late	Late	Early
Theory propagation	No	No	Yes
Used in Z3?	For bitvectors	No	Yes (CDCL(T))

Three Approaches Compared

	Eager	Lazy	DPLL(T)
Formula size	Huge	Original	Original
Theory consulted	Never	At the end	During search
Conflict detection	Late	Late	Early
Theory propagation	No	No	Yes
Used in Z3?	For bitvectors	No	Yes (CDCL(T))

DPLL(T) dominates. Early conflicts + theory propagation $\Rightarrow$ much less search. Architecture behind Z3, CVC5, Yices.

Theories in Action

Now: tour the major theories Z3 supports.

Theories in Action

Now: tour the major theories Z3 supports.

1. **LIA** (Linear Integer Arithmetic): SEND + MORE = MONEY.

Theories in Action

Now: tour the major theories Z3 supports.

1. **LIA** (Linear Integer Arithmetic): SEND + MORE = MONEY.
2. **EUF** (Equality + Uninterpreted Functions): function call reasoning.

Theories in Action

Now: tour the major theories Z3 supports.

1. **LIA** (Linear Integer Arithmetic): SEND + MORE = MONEY.
2. **EUF** (Equality + Uninterpreted Functions): function call reasoning.
3. **Arrays**: read/write reasoning.

LIA: SEND + MORE = MONEY

Classic cryptarithmetic. Each letter = distinct digit 0–9.

$$\texttt{SEND} + \texttt{MORE} = \texttt{MONEY}$$

LIA: SEND + MORE = MONEY

Classic cryptarithmetic. Each letter = distinct digit 0–9.

$$\texttt{SEND} + \texttt{MORE} = \texttt{MONEY}$$

$$1000S+100E+10N+D+1000M+100O+10R+E = 10000M+1000O+100N+10E+$$

LIA: SEND + MORE = MONEY

Classic cryptarithmetic. Each letter = distinct digit 0–9.

$$\texttt{SEND} + \texttt{MORE} = \texttt{MONEY}$$

$$1000S+100E+10N+D+1000M+100O+10R+E = 10000M+1000O+100N+10E+$$

Plus: $S \neq 0$, $M \neq 0$ (no leading zeros), all digits distinct. Pure LIA.

SEND + MORE = MONEY: Z3 Code

```
from z3 import *

S, E, N, D, M, O, R, Y = Ints('S E N D M O R Y')
s = Solver()

digits = [S, E, N, D, M, O, R, Y]
s.add([And(d >= 0, d <= 9) for d in digits])   # each digit 0-9
s.add(Distinct(digits))                         # all different
s.add(S != 0, M != 0)                          # no leading zeros
```

SEND + MORE = MONEY: Z3 Code

```
from z3 import *

S, E, N, D, M, O, R, Y = Ints('S E N D M O R Y')
s = Solver()

digits = [S, E, N, D, M, O, R, Y]
s.add([And(d >= 0, d <= 9) for d in digits])   # each digit 0-9
s.add(Distinct(digits))                         # all different
s.add(S != 0, M != 0)                          # no leading zeros

# The arithmetic constraint                    #New!!
send  = 1000*S + 100*E + 10*N + D
more  = 1000*M + 100*O + 10*R + E
money = 10000*M + 1000*O + 100*N + 10*E + Y

s.add(send + more == money)
```

SEND + MORE = MONEY: Z3 Code

```
from z3 import *

S, E, N, D, M, O, R, Y = Ints('S E N D M O R Y')
s = Solver()

digits = [S, E, N, D, M, O, R, Y]
s.add([And(d >= 0, d <= 9) for d in digits])   # each digit 0-9
s.add(Distinct(digits))                         # all different
s.add(S != 0, M != 0)                          # no leading zeros

# The arithmetic constraint                    #New!!
send  = 1000*S + 100*E + 10*N + D
more  = 1000*M + 100*O + 10*R + E
money = 10000*M + 1000*O + 100*N + 10*E + Y

s.add(send + more == money)

print(s.check())    # sat
m = s.model()
for v in digits:
    print(f"{v} = {m[v]}", end="  ")
# S = 9  E = 5  N = 6  D = 7  M = 1  O = 0  R = 8  Y = 2
# 9567 + 1085 = 10652
```

EUF: Reasoning About Function Calls

If $f(x) = y$ and $x = z$, must $f(z) = y$?

EUF: Reasoning About Function Calls

If $f(x) = y$ and $x = z$, must $f(z) = y$?

```
from z3 import *

x, y, z = Ints('x y z')
f = Function('f', IntSort(), IntSort())   #New!! uninterpreted function

s = Solver()
s.add(f(x) == y)
s.add(x == z)
s.add(f(z) != y)   # assert the negation: can f(z) differ from y?
```

EUF: Reasoning About Function Calls

If $f(x) = y$ and $x = z$, must $f(z) = y$?

```
from z3 import *

x, y, z = Ints('x y z')
f = Function('f', IntSort(), IntSort())   #New!! uninterpreted function

s = Solver()
s.add(f(x) == y)
s.add(x == z)
s.add(f(z) != y)   # assert the negation: can f(z) differ from y?
print(s.check())    # unsat
# No! By congruence: x = z implies f(x) = f(z).
# Since f(x) = y, we must have f(z) = y.
```

EUF: Reasoning About Function Calls

If $f(x) = y$ and $x = z$, must $f(z) = y$?

```
from z3 import *

x, y, z = Ints('x y z')
f = Function('f', IntSort(), IntSort())   #New!! uninterpreted function

s = Solver()
s.add(f(x) == y)
s.add(x == z)
s.add(f(z) != y)   # assert the negation: can f(z) differ from y?
print(s.check())    # unsat
# No! By congruence: x = z implies f(x) = f(z).
# Since f(x) = y, we must have f(z) = y.
```

We never defined f. Z3 used only the congruence axiom: EUF solver at work.

Arrays: Store and Select

Two operations:

- `Store(a, i, v)`: new array with $a[i] = v$.

Arrays: Store and Select

Two operations:

- `Store(a, i, v)`: new array with $a[i] = v$.
- `Select(a, i)`: read $a[i]$.

Arrays: Store and Select

Two operations:

- `Store(a, i, v)`: new array with $a[i] = v$.
- `Select(a, i)`: read $a[i]$.

Key axiom: $\text{Select}(\text{Store}(a, i, v), i) = v$.

Arrays: Store and Select

Two operations:

- `Store(a, i, v)`: new array with $a[i] = v$.
- `Select(a, i)`: read $a[i]$.

Key axiom: $\text{Select}(\text{Store}(a, i, v), i) = v$.

```
from z3 import *

a = Array('a', IntSort(), IntSort())   #New!! array: int -> int

# Write 42 at index 3, then read index 3
b = Store(a, 3, 42)
print(simplify(Select(b, 3)))   # 42
```

Arrays: Store and Select

Two operations:

- `Store(a, i, v)`: new array with $a[i] = v$.
- `Select(a, i)`: read $a[i]$.

Key axiom: $\text{Select}(\text{Store}(a, i, v), i) = v$.

```
from z3 import *

a = Array('a', IntSort(), IntSort())   #New!! array: int -> int

# Write 42 at index 3, then read index 3
b = Store(a, 3, 42)
print(simplify(Select(b, 3)))   # 42
# Harder: find an array where a[0] + a[1] + a[2] = 10
# and all entries are between 1 and 5
s = Solver()
for i in range(3):
    s.add(Select(a, i) >= 1, Select(a, i) <= 5)
s.add(Select(a, 0) + Select(a, 1) + Select(a, 2) == 10)

print(s.check())    # sat
print(s.model())    # e.g., a[0]=5, a[1]=4, a[2]=1
```

Why So Many Theories?

Why not one universal theory?

Why So Many Theories?

Why not one universal theory?

Each theory has a **specialized solver**:

- LIA: simplex + branch-and-bound.

Why So Many Theories?

Why not one universal theory?

Each theory has a **specialized solver**:

- LIA: simplex + branch-and-bound.
- EUF: union-find.

Why So Many Theories?

Why not one universal theory?

Each theory has a **specialized solver**:

- LIA: simplex + branch-and-bound.
- EUF: union-find.
- Arrays: axiom instantiation (read-over-write).

Why So Many Theories?

Why not one universal theory?

Each theory has a **specialized solver**:

- LIA: simplex + branch-and-bound.
- EUF: union-find.
- Arrays: axiom instantiation (read-over-write).
- Bitvectors: bit-blasting to SAT.

Why So Many Theories?

Why not one universal theory?

Each theory has a **specialized solver**:

- LIA: simplex + branch-and-bound.
- EUF: union-find.
- Arrays: axiom instantiation (read-over-write).
- Bitvectors: bit-blasting to SAT.

Same principle as Gurobi using different algorithms for LP vs. QP vs. MIP.

Why So Many Theories?

Why not one universal theory?

Each theory has a **specialized solver**:

- LIA: simplex + branch-and-bound.
- EUF: union-find.
- Arrays: axiom instantiation (read-over-write).
- Bitvectors: bit-blasting to SAT.

Same principle as Gurobi using different algorithms for LP vs. QP vs. MIP.

Specialization wins.

What If Your Formula Mixes Theories?

$$f(x) = f(y) \quad \wedge \quad x + 1 = y \quad \wedge \quad x > 0 \quad \wedge \quad \text{Select}(a, x) \neq \text{Select}(a, y)$$

What If Your Formula Mixes Theories?

$$f(x) = f(y) \quad \wedge \quad x + 1 = y \quad \wedge \quad x > 0 \quad \wedge \quad \text{Select}(a, x) \neq \text{Select}(a, y)$$

Mixes:

- **EUF**: $f(x) = f(y)$

What If Your Formula Mixes Theories?

$$f(x) = f(y) \quad \wedge \quad x + 1 = y \quad \wedge \quad x > 0 \quad \wedge \quad \text{Select}(a, x) \neq \text{Select}(a, y)$$

Mixes:

- **EUF**: $f(x) = f(y)$
- **LIA**: $x + 1 = y$, $x > 0$

What If Your Formula Mixes Theories?

$$f(x) = f(y) \quad \wedge \quad x + 1 = y \quad \wedge \quad x > 0 \quad \wedge \quad \text{Select}(a, x) \neq \text{Select}(a, y)$$

Mixes:

- **EUF**: $f(x) = f(y)$
- **LIA**: $x + 1 = y$, $x > 0$
- **Arrays**: $\text{Select}(a, x) \neq \text{Select}(a, y)$

What If Your Formula Mixes Theories?

$$f(x) = f(y) \quad \wedge \quad x + 1 = y \quad \wedge \quad x > 0 \quad \wedge \quad \text{Select}(a, x) \neq \text{Select}(a, y)$$

Mixes:

- **EUF**: $f(x) = f(y)$
- **LIA**: $x + 1 = y$, $x > 0$
- **Arrays**: $\text{Select}(a, x) \neq \text{Select}(a, y)$

No single solver handles all of this. How do we combine them?

Nelson-Oppen Combination (High Level)

Standard framework: **Nelson-Oppen** (1979).

Nelson-Oppen Combination (High Level)

Standard framework: **Nelson-Oppen** (1979).

1. **Purify:** split formula so each piece belongs to one theory.

Nelson-Oppen Combination (High Level)

Standard framework: **Nelson-Oppen** (1979).

1. **Purify:** split formula so each piece belongs to one theory.
2. **Solve independently:** each solver works on its piece.

Nelson-Oppen Combination (High Level)

Standard framework: **Nelson-Oppen** (1979).

1. **Purify:** split formula so each piece belongs to one theory.
2. **Solve independently:** each solver works on its piece.
3. **Share equalities:** solvers communicate equalities over shared variables.

Nelson-Oppen Combination (High Level)

Standard framework: **Nelson-Oppen** (1979).

1. **Purify:** split formula so each piece belongs to one theory.
2. **Solve independently:** each solver works on its piece.
3. **Share equalities:** solvers communicate equalities over shared variables.

Example: $x + 1 = y$ (LIA) and $f(x) = f(y)$ (EUF):

- LIA deduces $x \neq y$, shares with EUF.

Nelson-Oppen Combination (High Level)

Standard framework: **Nelson-Oppen** (1979).

1. **Purify:** split formula so each piece belongs to one theory.
2. **Solve independently:** each solver works on its piece.
3. **Share equalities:** solvers communicate equalities over shared variables.

Example: $x + 1 = y$ (LIA) and $f(x) = f(y)$ (EUF):

- LIA deduces $x \neq y$, shares with EUF.
- EUF: $f(x) = f(y)$ still consistent (f could map different inputs to same output). No contradiction from EUF alone.

Theory Combination in Practice

Z3 handles theory combination **automatically**.

Theory Combination in Practice

Z3 handles theory combination **automatically**.

You mix any theories. Z3 figures out:

- Which solver handles which atoms.

Theory Combination in Practice

Z3 handles theory combination **automatically**.

You mix any theories. Z3 figures out:

- Which solver handles which atoms.
- How to route shared variables.

Theory Combination in Practice

Z3 handles theory combination **automatically**.

You mix any theories. Z3 figures out:

- Which solver handles which atoms.
- How to route shared variables.
- How to combine answers.

Theory Combination in Practice

Z3 handles theory combination **automatically**.

You mix any theories. Z3 figures out:

- Which solver handles which atoms.
- How to route shared variables.
- How to combine answers.

Takeaway

Z3 = team of specialists, coordinated by a SAT engine. You just state the problem.