Programm- & Systemverifikation

SMT solvers

Georg Weissenbacher

(based on slides from Josef Widder)

184.741

In this talk

- What is SMT?
- Theories
  - equality logic
  - uninterpreted functions
  - linear arithmetic
- Solving simple SMT instances
  - removing constants
  - checking equality logic
  - reducing uninterpreted functions to equality logic
- How SMT deals with propositional structure
- Example
  - solver Z3
  - `http://rise4fun.com/z3`
  - `https://github.com/Z3Prover/z3`

What is SMT?

recall SAT:

- given a Boolean formula, e.g., $(\neg a \vee \neg b \vee c) \wedge (\neg a \vee b \vee d \vee e)$
- is there an assignment of true and false to variables a, b, c, d, e such that the formula evaluates to true?

What is SMT?

recall SAT:

- given a Boolean formula, e.g., $(\neg a \vee \neg b \vee c) \wedge (\neg a \vee b \vee d \vee e)$
- is there an assignment of true and false to variables a, b, c, d, e such that the formula evaluates to true?

Satisfiability Modulo Theories (SMT):

- given a formula, e.g.,

  $$x = y \;\wedge\; y = z \wedge u \neq x \wedge P(x, G(y, z)) \wedge G(y, z) = G(x, u)$$

  with

  - equality
  - functions such as G
  - predicates such as P
- is there an assignment of values to u, x, y, z such that formula evaluates to true?

Example theories we discuss in this lecture

- Equality logic:
  - $x = y \;\wedge\; y = z \;\wedge\; u \neq x \;\wedge\; z = u$
  - variables are of arbitrary domain (e.g., integers, reals, strings)

Example theories we discuss in this lecture

- Equality logic:
  - $x = y \ \wedge \ y = z \wedge u \neq x \wedge z = u$
  - variables are of arbitrary domain (e.g., integers, reals, strings)

- Equality logic with *uninterpreted functions*
  - $x = y \ \wedge \ y = z \wedge u \neq x \wedge z = G(x, u) \wedge G(y, z) = G(x, u)$
  - variables of arbitrary domain, and functions are unrestricted

Example theories we discuss in this lecture

- Equality logic:
  - $x = y \;\wedge\; y = z \wedge u \neq x \wedge z = u$
  - variables are of arbitrary domain (e.g., integers, reals, strings)
- Equality logic with *uninterpreted functions*
  - $x = y \;\wedge\; y = z \wedge u \neq x \wedge z = G(x, u) \wedge G(y, z) = G(x, u)$
  - variables of arbitrary domain, and functions are unrestricted
- (Linear) arithmetic
  - $(x + y \leq 1 \wedge 2x + y = 1) \vee 3x + 2y \geq 3$
  - variables are numbers
  - symbols have the standard interpretation of arithmetic

Other theories

- Arithmetic in general
  - e.g., $(x \cdot y \leq 1 \;\wedge\; 2x + y = 1) \;\vee\; y^2 \geq 3$

Other theories

- Arithmetic in general
  - e.g., $(x \cdot y \leq 1 \;\wedge\; 2x + y = 1) \;\vee\; y^2 \geq 3$
- Bit vectors
  - reduces essentially to SAT

Other theories

- Arithmetic in general
  - e.g., $(x \cdot y \leq 1 \ \wedge \ 2x + y = 1) \ \vee \ y^2 \geq 3$
- Bit vectors
  - reduces essentially to SAT
- Quantifiers (QBF)
  - $\forall x \exists y.\ x + y = 0$

Other theories

- Arithmetic in general
  - e.g., $(x \cdot y \leq 1 \ \wedge \ 2x + y = 1) \ \vee \ y^2 \geq 3$
- Bit vectors
  - reduces essentially to SAT
- Quantifiers (QBF)
  - $\forall x \exists y.\ x + y = 0$

…for details: Kroening, Strichman. Decision Procedures. Springer Verlag.

SMT and software engineering

C code fragment

```
int n = input();
int x = input();

int m = n;
int y = x;
int z = 0;

assume(n >= 0);

/* loop invariant:
   m * x == z + n * y */

while (n > 0) {
  if (n % 2) {
     z += y;
  }
  y *= 2;
  n /= 2;
}
assert (z == m * x);
```

C code fragment

```
int n = input();
int x = input();

int m = n;
int y = x;
int z = 0;

assume(n >= 0);

/* loop invariant:
   m * x == z + n * y */

while (n > 0) {
  if (n % 2) {
     z += y;
  }
  y *= 2;
  n /= 2;
}
assert (z == m * x);
```

encoding in Z3 (`loop.smt`)

SMT Encoding: Declare Variables

- Declare all program variables as SMT constants:

```
(declare-const n Int)
(declare-const x Int)
(declare-const m Int)
(declare-const y Int)
(declare-const z Int)
```

- ...and copies for variables that are modified:

```
(declare-const n2 Int)
(declare-const y2 Int)
(declare-const z2 Int)
```

SMT Encoding: Loop Body

- Define transition relation for loop body:

```
(define-fun loopcond () Bool (> n 0))

(define-fun loopbody () Bool
  (if loopcond
      (and (if (= 1 (mod n 2))
             (= z2 (+ z y))
           (= z2 z))
         (= y2 (* y 2))
         (= n2 (/ n 2)))
       (and (= z2 z)
      (= y2 y)
      (= n2 n))))
```

SMT Encoding: Loop Invariant

- Now we'd like to check our inductive loop invariant
- Let's define it first; wee need a copy for before the loop:

```
(define-fun invariant () Bool (and
              (>= n 0)
              (>= m 0)
              (= (* m x) (+ z (* n y)))))
```

- . . . and for afterwards:

```
(define-fun invariantpost () Bool (and
            (>= n2 0)
            (>= m 0)
            (= (* m x) (+ z2 (* n2 y2)))))
```

SMT Encoding: Base Case

- Let's check whether the precondition implies the invariant:

```
(push)
(assert (not (=>
      (and (= m n) (= x y) (= z 0) (>= n 0) )
      invariant
      )))
(check-sat)
(pop)
```

SMT Encoding: Induction Step

- Let's check whether consecution holds:

```
(push)
(assert (not (=>
        (and invariant loopbody)
        invariantpost)))
  (check-sat)
(pop)
```

SMT Encoding: Check Property

- Does the invariant imply the property?

```
(push)
  (assert (not (=>
    (and invariant (not loopcond))
    (= z (* m x)))))

  (check-sat)
(pop)
```

Why can’t we do that in SAT?

Why can't we do that in SAT?

If size of integers is fixed

- we can use boolean representation
  (recall bit-blasting from a previous lecture)

Why can’t we do that in SAT?

If size of integers is fixed

- we can use boolean representation
  (recall bit-blasting from a previous lecture)

If bit precision of integers is not fixed

- required to reason about arithmetic in general
- for certain data types, decision procedure can use specifics

Why can't we do that in SAT?

If size of integers is fixed

- we can use boolean representation
  (recall bit-blasting from a previous lecture)

If bit precision of integers is not fixed

- required to reason about arithmetic in general
- for certain data types, decision procedure can use specifics

Alert:

- if code should run on fixed-size integers
  then verification should not be done for general arithmetic:

$$a > b + 2 \wedge a \leq b \qquad \{a \mapsto 2, b \mapsto 2\}$$

(if we assume 2-bit integers)

Simple decision procedures

Equality logic

logical connectives	$\wedge, \vee, \neg$
atoms	$\mathit{term} = \mathit{term}$
term	variable name, or constant
domain	can be reals, integers, etc.

Equality logic — replace constants

- replace constants by variables
- add constraints imposed by the inequality of distinct constants e.g., $4 \neq 5$

Equality logic — replace constants

- replace constants by variables
- add constraints imposed by the inequality of distinct constants e.g., $4 \neq 5$

E.g. $x_1 = x_2 \;\wedge\; x_1 = x_3 \;\wedge\; x_1 = 5 \;\wedge\; x_2 = 4 \;\wedge\; x_3 = 5$

Equality logic — replace constants

- replace constants by variables
- add constraints imposed by the inequality of distinct constants
  e.g., $4 \neq 5$

$$\text{E.g. } x_1 = x_2 \ \wedge \ x_1 = x_3 \ \wedge \ x_1 = 5 \ \wedge \ x_2 = 4 \ \wedge \ x_3 = 5$$

- replace each constant C_i with a variable c_i
  e.g. replace 5 with c_1 and 4 with c_2

$$x_1 = x_2 \ \wedge \ x_1 = x_3 \ \wedge \ x_1 = c_1 \ \wedge \ x_2 = c_2 \ \wedge \ x_3 = c_1$$

Equality logic — replace constants

- replace constants by variables
- add constraints imposed by the inequality of distinct constants
  e.g., $4 \neq 5$

E.g. $x_1 = x_2 \;\wedge\; x_1 = x_3 \;\wedge\; x_1 = 5 \;\wedge\; x_2 = 4 \;\wedge\; x_3 = 5$

- replace each constant C_i with a variable c_i
  e.g. replace 5 with c_1 and 4 with c_2

$$x_1 = x_2 \;\wedge\; x_1 = x_3 \;\wedge\; x_1 = c_1 \;\wedge\; x_2 = c_2 \;\wedge\; x_3 = c_1$$

- for each pair of constants C_i and C_j with $i \neq j$ add $c_i \neq c_j$

$$x_1 = x_2 \wedge x_1 = x_3 \wedge x_1 = c_1 \wedge x_2 = c_2 \wedge x_3 = c_1 \wedge c_1 \neq c_2$$

Equality logic — check satisfiability (cont.)

$$x_1 = x_2 \;\wedge\; x_1 = x_3 \;\wedge\; x_1 = c_1 \;\wedge\; x_2 = c_2 \;\wedge\; x_3 = c_1 \;\wedge\; c_1 \neq c_2$$

Using equivalence classes:

$$\{x_1, x_2\}, \{x_1, x_3\}, \{x_1, c_1\}, \{x_2, c_2\}, \{x_3, c_1\}$$

Equality logic — check satisfiability (cont.)

$$x_1 = x_2 \ \wedge \ x_1 = x_3 \ \wedge \ x_1 = c_1 \ \wedge \ x_2 = c_2 \ \wedge \ x_3 = c_1 \ \wedge \ c_1 \neq c_2$$

Using equivalence classes:

$$\{x_1, x_2\}, \{x_1, x_3\}, \{x_1, c_1\}, \{x_2, c_2\}, \{x_3, c_1\}$$

Step 1: merge equivalence classes with shared term

Equality logic — check satisfiability (cont.)

$$x_1 = x_2 \ \wedge \ x_1 = x_3 \ \wedge \ x_1 = c_1 \ \wedge \ x_2 = c_2 \ \wedge \ x_3 = c_1 \ \wedge \ c_1 \neq c_2$$

Using equivalence classes:

$$\{x_1, x_2\}, \{x_1, x_3\}, \{x_1, c_1\}, \{x_2, c_2\}, \{x_3, c_1\}$$

Step 1: merge equivalence classes with shared term
$\{x_1, x_2, x_3\}, \{x_1, c_1\}, \{x_2, c_2\}, \{x_3, c_1\}$

Equality logic — check satisfiability (cont.)

$$x_1 = x_2 \ \wedge \ x_1 = x_3 \ \wedge \ x_1 = c_1 \ \wedge \ x_2 = c_2 \ \wedge \ x_3 = c_1 \ \wedge \ c_1 \neq c_2$$

Using equivalence classes:

$$\{x_1, x_2\}, \{x_1, x_3\}, \{x_1, c_1\}, \{x_2, c_2\}, \{x_3, c_1\}$$

Step 1: merge equivalence classes with shared term
$\{x_1, x_2, x_3\}, \{x_1, c_1\}, \{x_2, c_2\}, \{x_3, c_1\}$
$\{x_1, x_2, x_3, c_1\}, \{x_2, c_2\}, \{x_3, c_1\}$

Equality logic — check satisfiability (cont.)

$$x_1 = x_2 \ \wedge \ x_1 = x_3 \ \wedge \ x_1 = c_1 \ \wedge \ x_2 = c_2 \ \wedge \ x_3 = c_1 \ \wedge \ c_1 \neq c_2$$

Using equivalence classes:

$$\{x_1, x_2\}, \{x_1, x_3\}, \{x_1, c_1\}, \{x_2, c_2\}, \{x_3, c_1\}$$

Step 1: merge equivalence classes with shared term

$\{x_1, x_2, x_3\}, \{x_1, c_1\}, \{x_2, c_2\}, \{x_3, c_1\}$

$\{x_1, x_2, x_3, c_1\}, \{x_2, c_2\}, \{x_3, c_1\}$

$\{x_1, x_2, x_3, c_1, c_2\}, \{x_3, c_1\}$

Equality logic — check satisfiability (cont.)

$$x_1 = x_2 \ \wedge\ x_1 = x_3 \ \wedge\ x_1 = c_1 \ \wedge\ x_2 = c_2 \ \wedge\ x_3 = c_1 \ \wedge\ c_1 \neq c_2$$

Using equivalence classes:

$$\{x_1, x_2\}, \{x_1, x_3\}, \{x_1, c_1\}, \{x_2, c_2\}, \{x_3, c_1\}$$

Step 1: merge equivalence classes with shared term

$\{x_1, x_2, x_3\}, \{x_1, c_1\}, \{x_2, c_2\}, \{x_3, c_1\}$

$\{x_1, x_2, x_3, c_1\}, \{x_2, c_2\}, \{x_3, c_1\}$

$\{x_1, x_2, x_3, c_1, c_2\}, \{x_3, c_1\}$

$\{x_1, x_2, x_3, c_1, c_2\}$

Equality logic — check satisfiability (cont.)

$$x_1 = x_2 \;\wedge\; x_1 = x_3 \;\wedge\; x_1 = c_1 \;\wedge\; x_2 = c_2 \;\wedge\; x_3 = c_1 \;\wedge\; c_1 \neq c_2$$

Using equivalence classes:

$$\{x_1, x_2\}, \{x_1, x_3\}, \{x_1, c_1\}, \{x_2, c_2\}, \{x_3, c_1\}$$

Step 1: merge equivalence classes with shared term
$\{x_1, x_2, x_3\}, \{x_1, c_1\}, \{x_2, c_2\}, \{x_3, c_1\}$
$\{x_1, x_2, x_3, c_1\}, \{x_2, c_2\}, \{x_3, c_1\}$
$\{x_1, x_2, x_3, c_1, c_2\}, \{x_3, c_1\}$
$\{x_1, x_2, x_3, c_1, c_2\}$

Step 2: if there are two equivalent variables a, b, with $a \neq b$ in original formula return unsat else return sat

Equality logic — check satisfiability (cont.)

$$x_1 = x_2 \;\wedge\; x_1 = x_3 \;\wedge\; x_1 = c_1 \;\wedge\; x_2 = c_2 \;\wedge\; x_3 = c_1 \;\wedge\; c_1 \neq c_2$$

Using equivalence classes:

$$\{x_1, x_2\}, \{x_1, x_3\}, \{x_1, c_1\}, \{x_2, c_2\}, \{x_3, c_1\}$$

Step 1: merge equivalence classes with shared term
$\{x_1, x_2, x_3\}, \{x_1, c_1\}, \{x_2, c_2\}, \{x_3, c_1\}$
$\{x_1, x_2, x_3, c_1\}, \{x_2, c_2\}, \{x_3, c_1\}$
$\{x_1, x_2, x_3, c_1, c_2\}, \{x_3, c_1\}$
$\{x_1, x_2, x_3, c_1, c_2\}$

Step 2: if there are two equivalent variables a, b, with $a \neq b$ in original formula return unsat else return sat
e.g., since $c_1 \neq c_2$, unsat

Equality logic with uninterpreted functions EUF

logical connectives	$\wedge, \vee, \neg$
atoms	*term* $=$ *term*, predicate with parameters
term	variable name, or function symbol with parameters
domain	can be reals, integers, etc.

Example for EUF: equivalence of programs

```
x = (z * z) * z;
```

Example for EUF: equivalence of programs

```
x = (z * z) * z;
```

$$A \equiv x = F(F(z, z), z)$$

Example for EUF: equivalence of programs

```
x = (z * z) * z;
```

$$A \equiv x = F(F(z,z),z)$$

```
y = z;
y = y * z;
y = y * z;
```

Example for EUF: equivalence of programs

```
x = (z * z) * z;
```

$$A \equiv x = F(F(z,z),z)$$

```
y = z;
y = y * z;
y = y * z;
```

$$B \equiv y_0 = z \wedge y_1 = F(y_0, z) \wedge y_2 = F(y_1, z)$$

Example for EUF: equivalence of programs

```
x = (z * z) * z;
```

$$A \equiv x = F(F(z, z), z)$$

```
y = z;
y = y * z;
y = y * z;
```

$$B \equiv y_0 = z \wedge y_1 = F(y_0, z) \wedge y_2 = F(y_1, z)$$

program fragments equivalent if

$$A \wedge B \rightarrow x = y_2$$

Uninterpreted functions

Functional consistency. Instances of the same function return the same value if given equal arguments, that is, for all functions f:

$$\textit{if } x = y \textit{ then } f(x) = f(y)$$

Motivation

- check satisfiability of a formula ϕ that has a concrete function g
- replace g with uninterpreted function f to obtain ϕ^{UF}
- check validity of ϕ^{UF}.
  - if valid ϕ is valid
  - else: more refined analysis using g necessary

Other axioms can be added

- functional consistency is just *the* basic property
- if additional axioms are known, they can be added
  - commutativity $f(x, y) = f(y, x)$
  - associativity $f(f(x, y), z) = f(x, f(y, z))$
  - neutral element $x = f(x, 0)$

Other axioms can be added

- functional consistency is just *the* basic property
- if additional axioms are known, they can be added
  - commutativity $f(x, y) = f(y, x)$
  - associativity $f(f(x, y), z) = f(x, f(y, z))$
  - neutral element $x = f(x, 0)$
- Alert: the formula is growing larger…

Reducing EUF to equality logic

$$(x_1 \neq x_2) \ \lor \ (F(x_1) = F(x_2)) \ \lor \ (F(x_1) \neq F(x_3))$$

- idea: replace functions by variables
  - $F(x_1)$ with f_1, $F(x_2)$ with f_2, $F(x_3)$ with f_3

Reducing EUF to equality logic

$$(x_1 \neq x_2) \lor (F(x_1) = F(x_2)) \lor (F(x_1) \neq F(x_3))$$

- idea: replace functions by variables
  - $F(x_1)$ with f_1, $F(x_2)$ with f_2, $F(x_3)$ with f_3
- capture functional consistency constraints
  - $F(x_1) = F(x_2)$ must be true if $x_1 = x_2$
  - $F(x_1) \neq F(x_3)$ must be false if $x_1 = x_3$

Reducing EUF to equality logic (cont.)

$$(x_1 \neq x_2) \;\vee\; (F(x_1) = F(x_2)) \;\vee\; (F(x_1) \neq F(x_3))$$

Reducing EUF to equality logic (cont.)

$$(x_1 \neq x_2) \;\vee\; (F(x_1) = F(x_2)) \;\vee\; (F(x_1) \neq F(x_3))$$

functional constraints more general:

$$\begin{aligned} FC \;\equiv\; & (x_1 = x_2 \;\rightarrow\; f_1 = f_2) \;\wedge \\ & (x_1 = x_3 \;\rightarrow\; f_1 = f_3) \;\wedge \\ & (x_2 = x_3 \;\rightarrow\; f_2 = f_3) \end{aligned}$$

Reducing EUF to equality logic (cont.)

$$(x_1 \neq x_2) \;\vee\; (F(x_1) = F(x_2)) \;\vee\; (F(x_1) \neq F(x_3))$$

functional constraints more general:

$$\begin{aligned} FC \;\equiv\;\; & (x_1 = x_2 \;\rightarrow\; f_1 = f_2) \;\wedge \\ & (x_1 = x_3 \;\rightarrow\; f_1 = f_3) \;\wedge \\ & (x_2 = x_3 \;\rightarrow\; f_2 = f_3) \end{aligned}$$

flattening of function:

$$flat \equiv (x_1 \neq x_2) \;\vee\; (f_1 = f_2) \;\vee\; (f_1 \neq f_3)$$

Reducing EUF to equality logic (cont.)

$$(x_1 \neq x_2) \vee (F(x_1) = F(x_2)) \vee (F(x_1) \neq F(x_3))$$

functional constraints more general:

$$\begin{aligned} FC \quad \equiv \quad & (x_1 = x_2 \rightarrow f_1 = f_2) \wedge \\ & (x_1 = x_3 \rightarrow f_1 = f_3) \wedge \\ & (x_2 = x_3 \rightarrow f_2 = f_3) \end{aligned}$$

flattening of function:

$$\mathit{flat} \equiv (x_1 \neq x_2) \vee (f_1 = f_2) \vee (f_1 \neq f_3)$$

$FC \wedge \mathit{flat}$

- is in equality logic
- is valid if and only if the original formula is valid

Arithmetic

Linear Arithmetic — a decision procedure you know

consider a system of 3 equations with 2 variables

$$\begin{aligned} x + y &= 1 \\ 2x + y &= 1 \\ 3x + 2y &= 3 \end{aligned}$$

Linear Arithmetic — a decision procedure you know

consider a system of 3 equations with 2 variables

$$\begin{aligned} x + y &= 1 \\ 2x + y &= 1 \\ 3x + 2y &= 3 \end{aligned}$$

…Gaussian elimination

Linear Arithmetic — a decision procedure you know

consider a system of 3 equations with 2 variables

$$\begin{aligned} x + y &= 1 \\ 2x + y &= 1 \\ 3x + 2y &= 3 \end{aligned}$$

…Gaussian elimination

In other words, are there values for x and y satisfying

$$x + y = 1 \;\wedge\; 2x + y = 1 \;\wedge\; 3x + 2y = 3$$

geometric interpretation?

Linear Arithmetic — a decision procedure you know

consider a system of 3 equations with 2 variables

$$\begin{aligned} x + y &= 1 \\ 2x + y &= 1 \\ 3x + 2y &= 3 \end{aligned}$$

... Gaussian elimination

In other words, are there values for x and y satisfying

$$x + y = 1 \;\wedge\; 2x + y = 1 \;\wedge\; 3x + 2y = 3$$

geometric interpretation?

... but not only conjunctions

$$(x + y = 1 \;\wedge\; 2x + y = 1) \;\vee\; 3x + 2y = 3$$

Propositional Structure

How does SMT deal with disjunction?

- Decision procedues we encountered so far work for $\wedge$ only
- SMT cleverly combines SAT and theory reasoning:
  - SAT for efficient case splitting
  - Theory solvers for conjunctive reasoning

How SMT deals with disjunction

- SMT constructs a *propositional skeleton*; for

$$(\neg(\mathtt{x} = \mathtt{y}) \vee ((\mathtt{x}\&2) = 2)) \wedge (\mathtt{y} = \mathtt{z}+\mathtt{z}) \wedge (\mathtt{x} = \mathtt{z} \ll 1) \wedge ((\mathtt{z}\&1) = 0)$$

we get

$$(\neg e_1 \vee e_2) \wedge e_3 \wedge e_4 \wedge e_5 \,.$$

- Note: second formula is satisfiable, first one is not!

How SMT deals with disjunction (continued)

- Get satisfying assignment

  $$\{e_1 \mapsto 0, e_2 \mapsto 0, e_3 \mapsto 1, e_4 \mapsto 1, e_5 \mapsto 1\},$$

  encode it as conjunction:

  $$\neg e_1 \wedge \neg e_2 \wedge e_3 \wedge e_4 \wedge e_5$$

- Map back to original terms:

$$(x \neq y) \wedge ((x\&2) \neq 2) \wedge (y = z + z) \wedge (x = z \ll 1) \wedge ((z\&1) = 0)$$

How SMT deals with disjunction (continued)

- Now we can use a theory-specific solver for

$$(\mathtt{x} \neq \mathtt{y}) \wedge ((\mathtt{x}\&2) \neq 2) \wedge (\mathtt{y} = \mathtt{z}+\mathtt{z}) \wedge (\mathtt{x} = \mathtt{z} \ll 1) \wedge ((\mathtt{z}\&1) = 0)$$

How SMT deals with disjunction (continued)

- Now we can use a theory-specific solver for

$$(\mathtt{x} \neq \mathtt{y}) \wedge ((\mathtt{x}\&2) \neq 2) \wedge (\mathtt{y} = \mathtt{z}+\mathtt{z}) \wedge (\mathtt{x} = \mathtt{z} \ll 1) \wedge ((\mathtt{z}\&1) = 0)$$

$$\dfrac{\dfrac{\dfrac{\mathtt{x} = \mathtt{z} \ll 1 \qquad \mathtt{z} \ll 1 = \mathtt{z} + \mathtt{z}}{\mathtt{x} = \mathtt{z} + \mathtt{z}} \qquad \dfrac{\mathtt{y} = \mathtt{z} + \mathtt{z}}{\mathtt{z} + \mathtt{z} = \mathtt{y}}}{\mathtt{x} = \mathtt{y}} \qquad \mathtt{x} \neq \mathtt{y}}{\text{false}}$$

How SMT deals with disjunction (continued)

- Now we can use a theory-specific solver for

  $(\mathtt{x} \neq \mathtt{y}) \wedge ((\mathtt{x}\&2) \neq 2) \wedge (\mathtt{y} = \mathtt{z}{+}\mathtt{z}) \wedge (\mathtt{x} = \mathtt{z} \ll 1) \wedge ((\mathtt{z}\&1) = 0)$

$$\dfrac{\dfrac{\dfrac{\mathtt{x} = \mathtt{z} \ll 1 \qquad \mathtt{z} \ll 1 = \mathtt{z} + \mathtt{z}}{\mathtt{x} = \mathtt{z} + \mathtt{z}} \qquad \dfrac{\mathtt{y} = \mathtt{z} + \mathtt{z}}{\mathtt{z} + \mathtt{z} = \mathtt{y}}}{\mathtt{x} = \mathtt{y}} \qquad \mathtt{x} \neq \mathtt{y}}{\text{false}}$$

- Now we can block $(\mathtt{x} \neq \mathtt{y}) \wedge (\mathtt{y} = \mathtt{z} + \mathtt{z}) \wedge (\mathtt{x} = \mathtt{z} \ll 1)$ by adding clause $(e_1 \vee \neg e_3 \vee \neg e_4)$

How SMT deals with disjunction (continued)

- Now SAT solver continues with clauses

$$(\neg e_1 \vee e_2) \wedge e_3 \wedge e_4 \wedge e_5 \ \wedge \ (e_1 \vee \neg e_3 \vee \neg e_4)$$

How SMT deals with disjunction (continued)

- Now SAT solver continues with clauses

$$(\neg e_1 \vee e_2) \wedge e_3 \wedge e_4 \wedge e_5 \ \wedge \ (e_1 \vee \neg e_3 \vee \neg e_4)$$

- ... which is still satisfiable:

$$\{e_1 \mapsto 1, e_2 \mapsto 1, e_3 \mapsto 1, e_4 \mapsto 1, e_5 \mapsto 1\}\,,$$

How SMT deals with disjunction (continued)

- Now SAT solver continues with clauses

$$(\neg e_1 \vee e_2) \wedge e_3 \wedge e_4 \wedge e_5 \;\wedge\; (e_1 \vee \neg e_3 \vee \neg e_4)$$

- ...which is still satisfiable:

$$\{e_1 \mapsto 1, e_2 \mapsto 1, e_3 \mapsto 1, e_4 \mapsto 1, e_5 \mapsto 1\}\,,$$

- But we know that $e_2 \wedge e_4 \wedge e_5$ corresponds to

$$((\mathtt{x}\&2) \neq 2) \;\wedge\; (\mathtt{x} = \mathtt{z} \ll 1) \wedge ((\mathtt{z}\&1) = 0)$$

  which is unsatisfiable (using bit-vector arithmetic)

How SMT deals with disjunction (continued)

- We obtain:

$$
\begin{aligned}
&(\neg e_1 \lor e_2) \land e_3 \land e_4 \land e_5 \\
\land\ &(e_1 \lor \neg e_3 \lor \neg e_4) \\
\land\ &(\neg e_2 \lor \neg e_4 \lor \neg e_5)
\end{aligned}
$$

which is unsatisfiable (by unit propagation)!

How SMT deals with disjunction (continued)

- We obtain:

$$
\begin{aligned}
&(\neg e_1 \vee e_2) \wedge e_3 \wedge e_4 \wedge e_5 \\
\wedge\, &(e_1 \vee \neg e_3 \vee \neg e_4) \\
\wedge\, &(\neg e_2 \vee \neg e_4 \vee \neg e_5)
\end{aligned}
$$

  which is unsatisfiable (by unit propagation)!
- Hence, the original formula is unsatisfiable.

Things to take away

- sometimes applying SAT not possible
- closer to first order logic
  and sometimes beyond
- efficient procedures for specific theories
- extensive tool support
  - similar to SAT, there are competitions
  - agreed-upon input language `smtlib2`