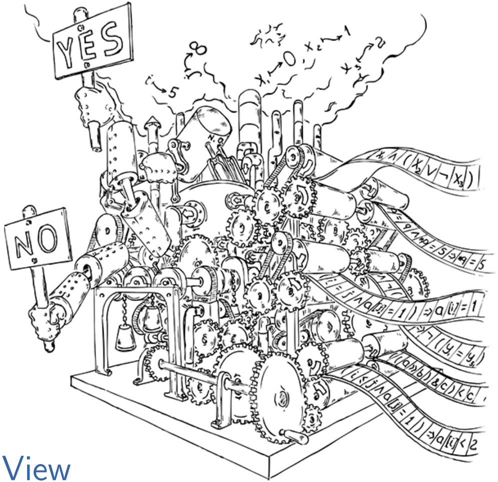
Bit-Vectors

Chapter 6



Decision Procedures

An Algorithmic Point of View

D.Kroening O.Strichman

Revision 1.2

Outline

1. Introduction to Bit-Vector Logic
2. Syntax
3. Semantics
4. Decision procedures for Bit-Vector Logic
   - Flattening Bit-Vector Logic
   - Incremental Flattening

What kind of logic do we need for system-level software?

```
State { int created = 0; }

IoCreateDevice.exit {
   if ($return==STATUS_SUCCESS)
     created = 1;
}

IoDeleteDevice.exit { created = 0; }

fun_AddDevice.exit {
   if (created && (pdevobj->Flags & DO_DEVICE_INITIALIZING) != 0) {
     abort "AddDevice routine failed to set "
           "~DO_DEVICE_INITIALIZING flag";
   }
}
```

An Invariant of Microsoft Windows Device Drivers

What kind of logic do we need for system-level software?

```
State { int created = 0; }

IoCreateDevice.exit {
   if ($return==STATUS_SUCCESS)
     created = 1;
}

IoDeleteDevice.exit { created = 0; }

fun_AddDevice.exit {
   if (created && (pdevobj->Flags & DO_DEVICE_INITIALIZING) != 0) {
     abort "AddDevice routine failed to set "
           "~DO_DEVICE_INITIALIZING flag";
   }
}
```

An Invariant of Microsoft Windows Device Drivers

What kind of logic do we need for system-level software?

```
State { int created = 0; }

IoCreateDevice.exit {
   if ($return==STATUS_SUCCESS)
     created = 1;
}

IoDeleteDevice.exit { created = 0; }

fun_AddDevice.exit {
   if (created && (pdevobj->Flags & DO_DEVICE_INITIALIZING) != 0) {
     abort "AddDevice routine failed to set "
           "~DO_DEVICE_INITIALIZING flag";
   }
}
```

Bit-wise AND

An Invariant of Microsoft Windows Device Drivers

What kind of logic do we need for system-level software?

- We need bit-vector logic – with bit-wise operators, arithmetic overflow
- We want to scale to large programs – must verify large formulas

What kind of logic do we need for system-level software?

- We need bit-vector logic – with bit-wise operators, arithmetic overflow
- We want to scale to large programs – must verify large formulas
- Examples of program analysis tools that generate bit-vector formulas:
  - CBMC
  - SATABS
  - F-Soft (NEC)
  - SATURN (Stanford, Alex Aiken)
  - EXE (Stanford, Dawson Engler, David Dill)
  - Variants of those developed at IBM, Microsoft

$$
\begin{array}{rcl}
\mathit{formula} & : & \mathit{formula} \lor \mathit{formula} \mid \neg \mathit{formula} \mid \mathit{atom} \\
\mathit{atom} & : & \mathit{term}\ \mathit{rel}\ \mathit{term} \mid \mathit{Boolean\text{-}Identifier} \mid \mathit{term}[\,\mathit{constant}\,] \\
\mathit{rel} & : & = \mid < \\
\mathit{term} & : & \mathit{term}\ \mathit{op}\ \mathit{term} \mid \mathit{identifier} \mid \sim \mathit{term} \mid \mathit{constant} \mid \\
 & & \mathit{atom}?\mathit{term}:\mathit{term} \mid \\
 & & \mathit{term}[\,\mathit{constant} : \mathit{constant}\,] \mid \boxed{\mathit{ext}(\,\mathit{term}\,)} \qquad ext_{[16]}(x_{[8]}) \\
\mathit{op} & : & + \mid - \mid \cdot \mid / \mid << \mid >> \mid \& \mid | \mid \oplus \mid \circ \quad = 0 \ldots 0\, x_7 \ldots x_0
\end{array}
$$

$y_\ell := x_\ell + 1$

if $x_0 = 0$ then $y_0 := 1$ $y_1 \ldots y_{\ell-1} := x_1 \ldots x_{\ell-1}$

else if $x_1 = 0$ then $y_1 := 1$ $y_0, y_2 \ldots y_{\ell-1} := x_0, x_2, \ldots x_{\ell-1}$

⋮

Bit-Vector Logic: Syntax

$$\begin{array}{rcl}
formula & : & formula \vee formula \mid \neg formula \mid atom \\
atom & : & term\ rel\ term \mid Boolean\text{-}Identifier \mid term[\,constant\,] \\
rel & : & = \ \mid\ < \\
term & : & term\ op\ term \mid identifier \mid \ \sim term \mid constant \mid \\
 & & atom ? term : term \mid \\
 & & term[\,constant : constant\,] \mid ext(\,term\,) \\
op & : & + \mid - \mid \cdot \mid / \mid \ll \mid \gg \mid \& \mid \mid \mid \oplus \mid \circ
\end{array}$$

- $\sim x$: bit-wise negation of x
- $ext(x)$: sign- or zero-extension of x
- $x \ll d$: left shift with distance d
- $x \circ y$: concatenation of x and y

Danger!

$$(x - y > 0) \iff (x > y)$$

Valid over $\mathbb{R}/\mathbb{N}$, but not over the bit-vectors.
(Many compilers have this sort of bug)

Width and Encoding

- The meaning depends on the width and encoding of the variables.

- The meaning depends on the width and encoding of the variables.
- Typical encodings:

$a_{[l]} = a_{l-1} \dots a_0$

  - Binary encoding

$$\langle x\rangle_U := \sum_{i=0}^{l-1} a_i \cdot 2^i$$

$a_{[4]U} = \langle 0101 \rangle_U =$
3. 2. 1. 0.
$= 2^2 + 2^0 = 5$

  - Two's complement

$$\langle x\rangle_S := -2^{\overset{l-1}{\cancel{n-1}}} \cdot a_{\underset{l-1}{\cancel{n-1}}} + \sum_{i=0}^{l-2} a_i \cdot 2^i$$

$a_{[4]S} = \langle \boxed{1}101 \rangle_S =$
4. 2. 1. 0.
$= -2^3 + 2^2 + 2^0 =$
$-8 + 4 + 1 = -3$

$a_{[4]S} = \langle 0101 \rangle_S =$
$2^2 + 2^0 = 5$

  - But maybe also fixed-point, $\boxed{\text{floating-point,}}$ ...

Examples

$2^7 + 2^6 + 2^3 = 128 + 64 + 8$

$$
\begin{aligned}
&\overset{7\,6\,5\,4\,3\,2\,1\,0}{\langle 11001000\rangle_U} &&= 200\\
&\langle 11001000\rangle_S &&= -128 + 64 + 8 = -56\\
&\langle 01100100\rangle_S &&= 100
\end{aligned}
$$

Width and Encoding

Notation to clarify width and encoding:

$$x_{[32]S}$$

Notation to clarify width and encoding:

$$x_{[32]S}$$

Width in bits

U: unsigned binary
S: signed two's complement

Bit-vectors Made Formal

Definition (Bit-Vector)

A *bit-vector* is a vector of Boolean values with a given length l:

$$b : \{0, \ldots, l-1\} \longrightarrow \{0, 1\}$$

Definition (Bit-Vector)

A *bit-vector* is a vector of Boolean values with a given length l:

$$b : \{0, \ldots, l-1\} \longrightarrow \{0, 1\}$$

The value of bit number i of x is $x(i)$.

$$\underbrace{\boxed{b_{l-1}}\boxed{b_{l-2}}\cdots\cdots\boxed{b_2}\boxed{b_1}\boxed{b_0}}_{l\ \text{bits}}$$

We also write x_i for $x(i)$.

λ expressions are functions without a name

λ expressions are functions without a name

Examples:

- The vector of length l that consists of zeros:

$$\lambda i \in \{0, \ldots, l-1\}.0$$

- A function that inverts (flips all bits in) a bit-vector:

$$\textit{bv-invert}(x) := \lambda i \in \{0, \ldots, l-1\}.\neg x_i$$

~ x

- A bit-wise OR:

$$\textit{bv-or}(x, y) := \lambda i \in \{0, \ldots, l-1\}.(x_i \vee y_i)$$

x | y

$\Longrightarrow$ we now have semantics for the bit-wise operators.

Example

$$(x_{[10]} \circ y_{[5]})[14] \iff x[9]$$

Example

$$(x_{[10]} \circ y_{[5]})[14] \iff x[9]$$

- This is translated as follows:

$$x[9] \quad = \quad x_9$$

Example

$$(x_{[10]} \circ y_{[5]})[14] \iff x[9]$$

- This is translated as follows:

$$x[9] \quad = \quad x_9$$

$$(x \circ y) \quad = \quad \lambda i.(i < 5) ? \underline{y_i} : \underline{x_{i-5}}$$

$(x \circ y)[5] = x[5-5] = x[0]$

Example

$$(x_{[10]} \circ y_{[5]})[14] \iff x[9]$$

- This is translated as follows:

$$x[9] \quad = \quad x_9$$

$$(x \circ y) \quad = \quad \lambda i.(i < 5)?y_i : x_{i-5}$$

$$(x \circ y)[14] \quad = \quad (\lambda i.(i < 5)?y_i : x_{i-5})(14)$$

Example

$$(x_{[10]} \circ y_{[5]})[14] \iff x[9]$$

- This is translated as follows:

$$x[9] \quad = \quad x_9$$

$$(x \circ y) \quad = \quad \lambda i.(i < 5)?y_i : x_{i-5}$$

$$(x \circ y)[14] \quad = \quad (\lambda i.(i < 5)?y_i : x_{i-5})(14)$$

- Final result:

$$(\lambda i.(i < 5)?y_i : x_{i-5})(14) \iff x_9$$

What is the output of the following program?

```
unsigned char number = 200;
number = number + 100;
printf("Sum: %d\n", number);
```

What is the output of the following program?

```
unsigned char number = 200;
number = number + 100;
printf("Sum: %d\n", number);
```

On most architectures, this is 44!

$$
\begin{array}{lrcl}
 & 11001000 & = & 200 \\
+_u & 01100100 & = & 100 \\
\hline
= & 00101100 & = & 44
\end{array}
$$

What is the output of the following program?

```
unsigned char number = 200;
number = number + 100;
printf("Sum: %d\n", number);
```

On most architectures, this is 44!

$$\begin{array}{ccc} & 11001000 & = 200 \\ + & 01100100 & = 100 \\ \hline = & 00101100 & = 44 \end{array}$$

$\Longrightarrow$ Bit-vector arithmetic uses modular arithmetic!

Semantics for addition, subtraction:

$$a_{[l]} +_U b_{[l]} = c_{[l]} \iff (\langle a \rangle_U + \langle b \rangle_U) = \langle c \rangle_U \mod 2^l$$

$$a_{[l]} -_U b_{[l]} = c_{[l]} \iff \langle a \rangle_U - \langle b \rangle_U = \langle c \rangle_U \mod 2^l$$

Semantics for addition, subtraction:

$$
\begin{aligned}
a_{[l]} +_U b_{[l]} = c_{[l]} \quad &\Longleftrightarrow \quad \langle a\rangle_U + \langle b\rangle_U = \langle c\rangle_U \mod 2^l \\
a_{[l]} -_U b_{[l]} = c_{[l]} \quad &\Longleftrightarrow \quad \langle a\rangle_U - \langle b\rangle_U = \langle c\rangle_U \mod 2^l \\
a_{[l]} +_S b_{[l]} = c_{[l]} \quad &\Longleftrightarrow \quad \langle a\rangle_S + \langle b\rangle_S = \langle c\rangle_S \mod 2^l \\
a_{[l]} -_S b_{[l]} = c_{[l]} \quad &\Longleftrightarrow \quad \langle a\rangle_S - \langle b\rangle_S = \langle c\rangle_S \mod 2^l
\end{aligned}
$$

Semantics for addition, subtraction:

$$
\begin{aligned}
a_{[l]} +_U b_{[l]} = c_{[l]} &\iff \langle a\rangle_U + \langle b\rangle_U = \langle c\rangle_U \mod 2^l \\
a_{[l]} -_U b_{[l]} = c_{[l]} &\iff \langle a\rangle_U - \langle b\rangle_U = \langle c\rangle_U \mod 2^l \\
a_{[l]} +_S b_{[l]} = c_{[l]} &\iff \langle a\rangle_S + \langle b\rangle_S = \langle c\rangle_S \mod 2^l \\
a_{[l]} -_S b_{[l]} = c_{[l]} &\iff \langle a\rangle_S - \langle b\rangle_S = \langle c\rangle_S \mod 2^l
\end{aligned}
$$

We can even mix the encodings:

$$
a_{[l]U} +_U b_{[l]S} = c_{[l]U} \iff \langle a\rangle_U + \langle b\rangle_S = \langle c\rangle_U \mod 2^l
$$

Semantics for Relational Operators

Semantics for $<$, $\leq$, $\geq$, and so on:

$$
\begin{aligned}
a_{[l]U} <_u b_{[l]U} &\iff \langle a \rangle_U < \langle b \rangle_U \\
a_{[l]S} <_s b_{[l]S} &\iff \langle a \rangle_S < \langle b \rangle_S
\end{aligned}
$$

Semantics for $<$, $\leq$, $\geq$, and so on:

$$\begin{aligned} a_{[l]U} < b_{[l]U} \quad &\iff \quad \langle a\rangle_U < \langle b\rangle_U \\ a_{[l]S} < b_{[l]S} \quad &\iff \quad \langle a\rangle_S < \langle b\rangle_S \end{aligned}$$

Mixed encodings:

$$\begin{aligned} a_{[l]U} < b_{[l]S} \quad &\iff \quad \langle a\rangle_U < \langle b\rangle_S \\ a_{[l]S} < b_{[l]U} \quad &\iff \quad \langle a\rangle_S < \langle b\rangle_U \end{aligned}$$

Note that most compilers don't support comparisons with mixed encodings.

Complexity

- Satisfiability is undecidable for an unbounded width, even without arithmetic.

Complexity

- Satisfiability is undecidable for an unbounded width, even without arithmetic.
- It is NP-complete otherwise.

A Simple Decision Procedure

- Transform Bit-Vector Logic to Propositional Logic
- Most commonly used decision procedure
- Also called '*bit-blasting*'

- Transform Bit-Vector Logic to Propositional Logic
- Most commonly used decision procedure
- Also called '*bit-blasting*'

$$((\overbrace{\overbrace{(a+c)}^{x_3}\cdot d}^{x_4}=f)\wedge(\underbrace{a=\overbrace{b+c}^{x_1}}_{e_1}\vee\underbrace{b=\overbrace{c\mid d}^{x_2}}_{e_2})\wedge\underbrace{x<y}_{e_3}$$

$$e_0\wedge(e_1\vee e_2)\wedge e_3$$

Bit-Vector Flattening

1. Convert propositional part as before e_i
2. Add a *Boolean variable for each bit* of each sub-expression (term) $e_i \Leftrightarrow$ „meaning of e_i" (e.g., $e_0 \Leftrightarrow x_4 = f$
3. Add *constraint* for each sub-expression $x_i =$ „meaning of x_i" (e.g., $x_3 = a + c$)

We denote the new Boolean variable for bit i of term t by $\mu(t)_i$.

$$\underbrace{a_{[e]} = b_{[e]}}_{e_1} \qquad \underbrace{a <_u b}_{e_2}$$

$$(\;\dots\; e_1 \dots e_2 \dots) \wedge \left(e_1 \Leftrightarrow \bigwedge_{i=0}^{e} a_i = b_i \right) \wedge$$

$$\begin{aligned} \Big(e_2 \Leftrightarrow \Big[& (a_3 = 0 \wedge b_3 = 1) \vee \\ & (a_3 = b_3 \wedge a_2 = 0 \wedge b_2 = 1) \vee \\ & (a_3 = b_3 \wedge a_2 = b_2 \wedge a_1 = 0 \wedge b_1 = 1) \vee \\ & (a_3 = b_3 \wedge a_2 = b_2 \wedge a_1 = b_1 \wedge \\ & \qquad a_0 = 0 \wedge b_0 = 1) \Big] \Big) \end{aligned}$$

a	0	0	1	1
b	0	1	1	0

$\mapsto$

What constraints do we generate for a given term?

What constraints do we generate for a given term?

- This is easy for the bit-wise operators.
- Example for $a|_{[l]}b$:

$$\bigwedge_{i=0}^{l-1}(\underbrace{\mu(t)_i}_{x} = (a_i \vee b_i))$$

(read $x = y$ over bits as $x \iff y$)

$$\underbrace{c = \overbrace{a |_{[4]} b}^{x}}_{e} \rightsquigarrow e \wedge c = x \wedge x_0 \Leftrightarrow (a_0 \vee b_0) \wedge x_1 \Leftrightarrow (a_1 \vee b_1) \wedge x_2 \Leftrightarrow (a_2 \vee b_2) \wedge x_3 \Leftrightarrow (a_3 \vee b_3)$$

What constraints do we generate for a given term?

- This is easy for the bit-wise operators.

- Example for $a|_{[l]}b$:

$$\bigwedge_{i=0}^{l-1}(\mu(t)_i = (a_i \vee b_i))$$

(read $x = y$ over bits as $x \iff y$)

- We can transform this into CNF using Tseitin's method.

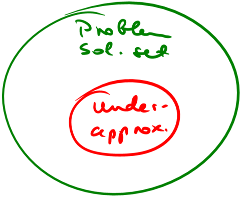
a, b, c bit-vectors of length 2

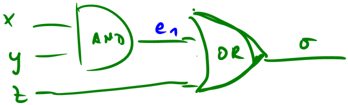
$$\underbrace{a = (\underbrace{b \circledast c}_{x})}_{e} \Leftrightarrow e \wedge$$

$$(e \Leftrightarrow (a_0 \Leftrightarrow x_0 \wedge a_1 \Leftrightarrow x_1)) \wedge$$

$$(x_0 \Leftrightarrow [(b_0 \vee c_0) \wedge (\neg b_0 \vee \neg c_0)]) \wedge$$

$$(x_1 \Leftrightarrow [(b_1 \vee c_1) \wedge (\neg b_1 \vee \neg c_1)])$$



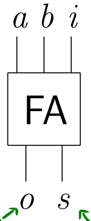


$$\big(\sigma \Leftrightarrow (e_1 \vee z)\big) \wedge$$
$$\big(e_1 \Leftrightarrow (x \wedge y)\big)$$

How to flatten $a + b$?

How to flatten $a + b$? (a, b are bits)

$\longrightarrow$ we can build a *circuit* that adds them!



overflow ↗ ↖ result bit

Full Adder

$$s \equiv (a + b + i) \bmod 2 \equiv a \oplus b \oplus i$$

a∧b ∨ a∧i ∨ b∧i

$$o \equiv (a + b + i) \operatorname{div} 2 \equiv \boxed{a \cdot b + a \cdot i + b \cdot i}$$

The full adder in CNF:

o: $(a \vee b \vee \neg o) \wedge (a \vee \neg b \vee i \vee \neg o) \wedge (a \vee \neg b \vee \neg i \vee o) \wedge$
$(\neg a \vee b \vee i \vee \neg o) \wedge (\neg a \vee b \vee \neg i \vee o) \wedge (\neg a \vee \neg b \vee o)$ 6 clauses

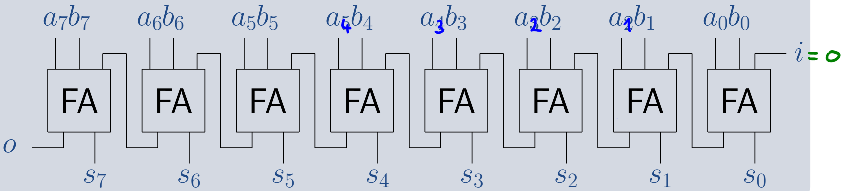
s: $(a \vee b \vee i \vee \neg s) \wedge (a \vee b \vee \neg i \vee s) \wedge (a \vee \neg b \vee i \vee s) \wedge$
$(a \vee b \vee \neg i \vee \neg s) \wedge (\neg a \vee b \vee i \vee s) \wedge (\neg a \vee b \vee \neg i \vee \neg s) \wedge$
$(\neg a \vee \neg b \vee i \vee \neg s) \wedge (\neg a \vee \neg b \vee \neg i \vee s)$ 8 clauses

14 clauses

Ok, this is good for one bit! How about more?

Ok, this is good for one bit! How about more?

8-Bit ripple carry adder (RCA) $a_{[8]u} +_u b_{[8]u} = s_{[8]u}$



- Also called *carry chain adder*
- Adds $2l$ variables
- Adds $\cancel{6}_{14} \cdot l$ clauses

Multipliers

- Multipliers result in very hard formulas
- Example:

$$a \cdot b = c \wedge b \cdot a \neq c \wedge x < y \wedge x > y$$

  CNF: About 11000 variables, unsolvable for current SAT solvers

- Similar problems with division, modulo
- Q: Why is this hard?

- Multipliers result in very hard formulas
- Example:

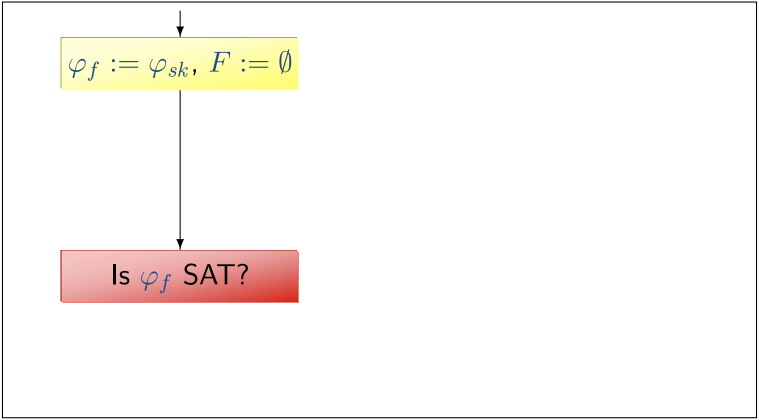
$$a \cdot b = c \wedge b \cdot a \neq c \wedge x < y \wedge x > y$$

CNF: About 11000 variables, unsolvable for current SAT solvers

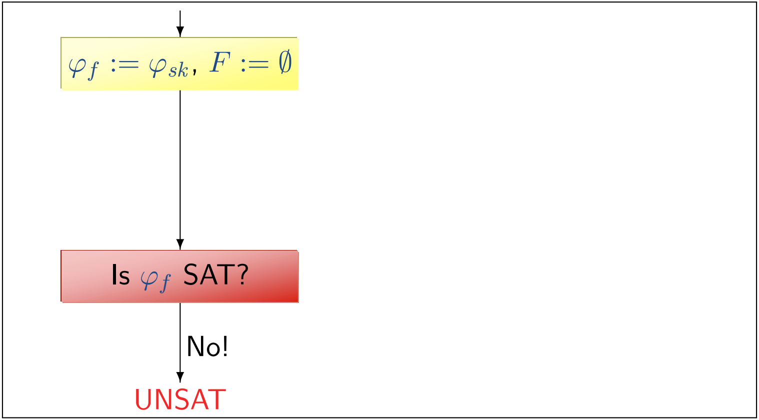
- Similar problems with division, modulo
- Q: Why is this hard?
- Q: How do we fix this?

$\varphi_f := \varphi_{sk},\ F := \emptyset$

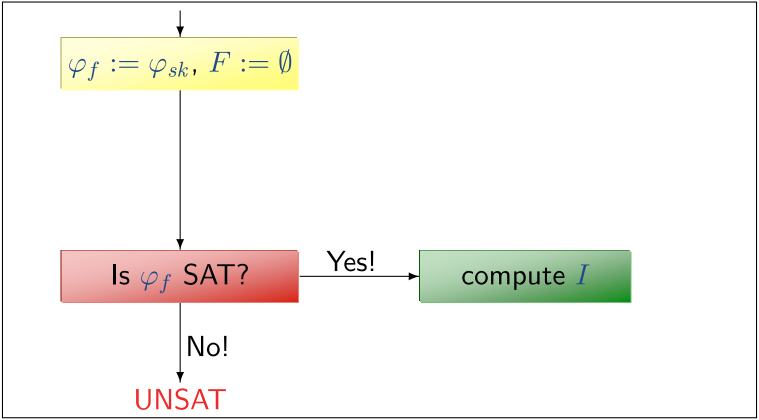
φ_{sk}: Boolean part of φ
F: set of terms that are in the encoding



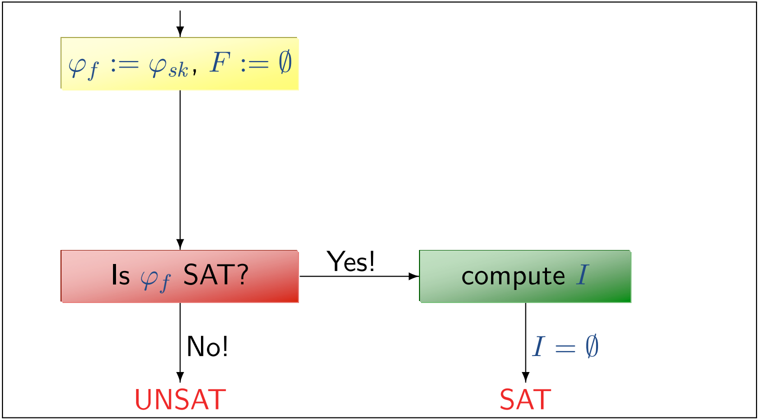
φ_{sk}: Boolean part of φ
F: set of terms that are in the encoding



φ_{sk}: Boolean part of φ
F: set of terms that are in the encoding

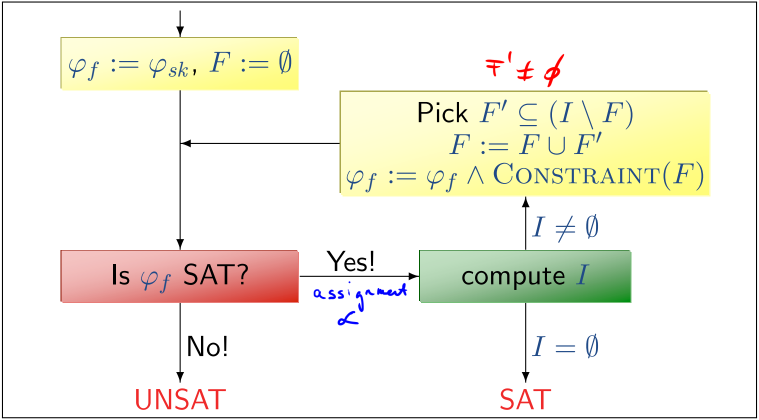


φ_{sk}: Boolean part of φ
F: set of terms that are in the encoding
I: set of terms that are inconsistent with the current assignment



φ_{sk}: Boolean part of φ
F: set of terms that are in the encoding
I: set of terms that are inconsistent with the current assignment

CEGAR : Counterexample-guided abstraction refinement



φ_{sk}: Boolean part of φ

F: set of terms that are in the encoding

I: set of terms that are inconsistent with the current assignment

Incremental Flattening

- Idea: add 'easy' parts of the formula first
- Only add hard parts when needed
- φ_f only gets stronger – use an incremental SAT solver

Incomplete Assignments

- Hey: initially, we only have the skeleton!
  How do we know what terms are inconsistent with the current assignment if the variables aren't even in φ_f?

Incomplete Assignments

- Hey: initially, we only have the skeleton!
  How do we know what terms are inconsistent with the current assignment if the variables aren't even in φ_f?

- Solution: guess some values for the missing variables.
  If you guess right, it's good.

- Hey: initially, we only have the skeleton!
  How do we know what terms are inconsistent with the current assignment if the variables aren't even in φ_f?

- Solution: guess some values for the missing variables.
  If you guess right, it's good.

- Ideas:
  - All zeros
  - Sign extension for signed bit-vectors
  - Try to propagate constants ($a = b + 1$)

$$\underbrace{\underbrace{a+b}_{z} = c}_{e_1} \wedge \underbrace{x < y}_{e_2} \wedge \underbrace{x > y}_{e_3}$$

$$e_1 \wedge e_2 \wedge e_3$$

$\rightarrow$ SAT $\alpha(e_i) = 1$

$\alpha(x_i) = \alpha(a) = \alpha(b) = \alpha(c) = 1$

violated : e_1, $\underline{e_2}$, $\underline{e_3}$

$$e_1 \wedge e_2 \wedge e_3 \wedge (\neg x_0 \wedge y_0) \rightarrow \text{SAT} \quad \alpha(e_i) = 1$$

$\alpha(y_0) = 1 \quad \alpha(x_0) = 0$

$\alpha(a) = \alpha(b) = \alpha(c) = 1$

violated : e_1 $\underline{\underline{e_3}}$

$$e_1 \wedge e_2 \wedge e_3 \wedge (\neg x_0 \wedge y_0) \wedge (x_0 \wedge \neg y_0) \Rightarrow \text{UNSAT}$$