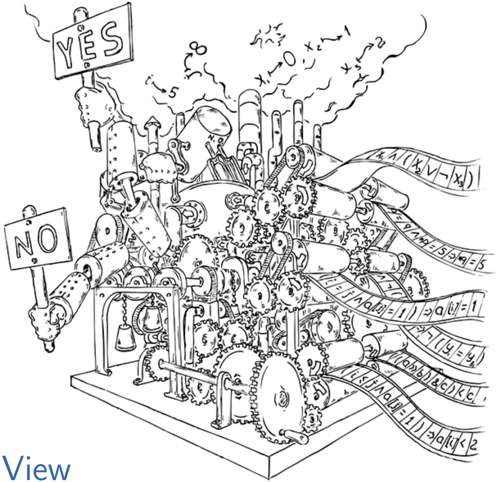
Bit-Vectors

Chapter 6



Decision Procedures

An Algorithmic Point of View

D.Kroening O.Strichman

Revision 1.2

Outline

Bit-Vector Logic: Syntax

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

Bit-Vector Logic: Syntax

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

- $\sim x$: 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.

Width and Encoding

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

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

  - Two's complement

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

  - But maybe also fixed-point, floating-point, ...

Examples

$$\langle 11001000\rangle_U = 200$$

$$\langle 11001000\rangle_S = -128 + 64 + 8 = -56$$

$$\langle 01100100\rangle_S = 100$$

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

Definition (Bit-Vector)

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

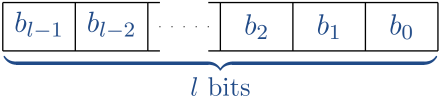
$$b : \{0, \ldots, l-1\} \longrightarrow \{0, 1\}$$

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 $x(i)$.



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

Lambda-Notation for Bit-Vectors

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:

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

- A bit-wise OR:

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

$\Longrightarrow$ 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:

$$x[9] = x_9$$

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

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:

$$x[9] = x_9$$

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

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

- 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!

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

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!

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

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

Semantics for Arithmetic Expressions

Semantics for addition, subtraction:

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

Semantics for addition, subtraction:

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

Semantics for addition, subtraction:

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

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

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

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

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

$$a_{[l]U} < b_{[l]U} \iff \langle a \rangle_U < \langle b \rangle_U$$
$$a_{[l]S} < b_{[l]S} \iff \langle a \rangle_S < \langle b \rangle_S$$

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

$$a_{[l]U} < b_{[l]U} \iff \langle a \rangle_U < \langle b \rangle_U$$
$$a_{[l]S} < b_{[l]S} \iff \langle a \rangle_S < \langle b \rangle_S$$

Mixed encodings:

$$a_{[l]U} < b_{[l]S} \iff \langle a \rangle_U < \langle b \rangle_S$$
$$a_{[l]S} < b_{[l]U} \iff \langle a \rangle_S < \langle b \rangle_U$$

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

Complexity

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

- 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*'

Bit-Vector Flattening

1. Convert propositional part as before
2. Add a *Boolean variable for each bit* of each sub-expression (term)
3. Add *constraint* for each sub-expression

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

Algorithm 6.2.1: BV-FLATTENING

Input: A formula φ in bit-vector arithmetic
Output: An equisatisfiable Boolean formula $\mathcal{B}$

function BV-FLATTENING
  $\mathcal{B} := e(\varphi)$; ▷ the propositional skeleton of φ
  for each $t_{[l]} \in T(\varphi)$ **do**
    for each $i \in \{0, \ldots, l-1\}$ **do**
      set $e(t)_i$ to a new Boolean variable;
  for each $a \in At(\varphi)$ **do**
    $\mathcal{B} := \mathcal{B} \wedge$ BV-CONSTRAINT(e, a);
  for each $t_{[l]} \in T(\varphi)$ **do**
    $\mathcal{B} := \mathcal{B} \wedge$ BV-CONSTRAINT(e, t);
  return $\mathcal{B}$;

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 $a|_{[l]}b$:

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

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

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 $x \iff y$)

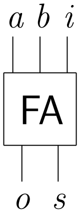
- We can transform this into CNF using Tseitin's method.

Flattening Bit-Vector Arithmetic

How to flatten $a + b$?

How to flatten $a + b$?

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



Full Adder

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

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

The full adder in CNF:

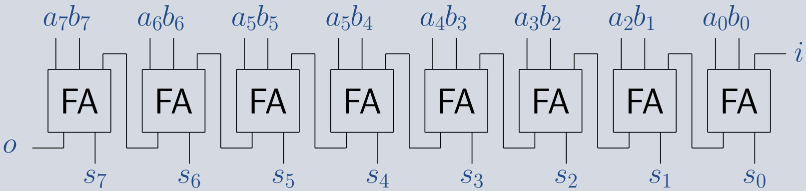
$$(a \lor b \lor \neg o) \land (a \lor \neg b \lor i \lor \neg o) \land (a \lor \neg b \lor \neg i \lor o) \land$$
$$(\neg a \lor b \lor i \lor \neg o) \land (\neg a \lor b \lor \neg i \lor o) \land (\neg a \lor \neg b \lor o)$$

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

Flattening Bit-Vector Arithmetic

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

8-Bit ripple carry adder (RCA)



- Also called *carry chain adder*
- Adds l variables
- Adds $6 \cdot l$ clauses

- Multipliers result in very hard formulas
- Example:

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

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

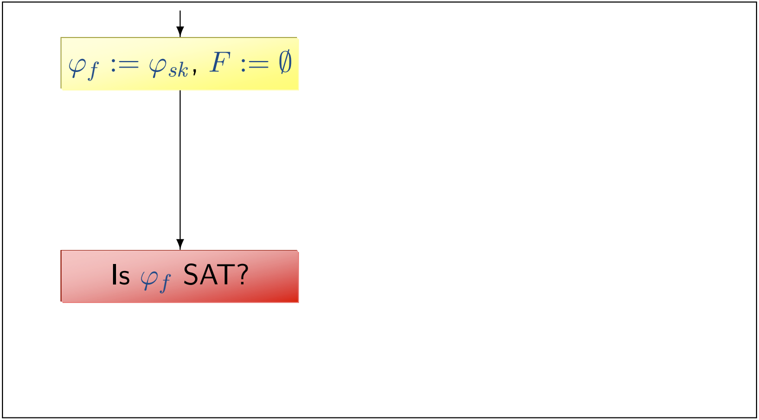
- Similar problems with division, modulo
- Q: Why is this hard?
- Q: How do we fix this?

Incremental Flattening

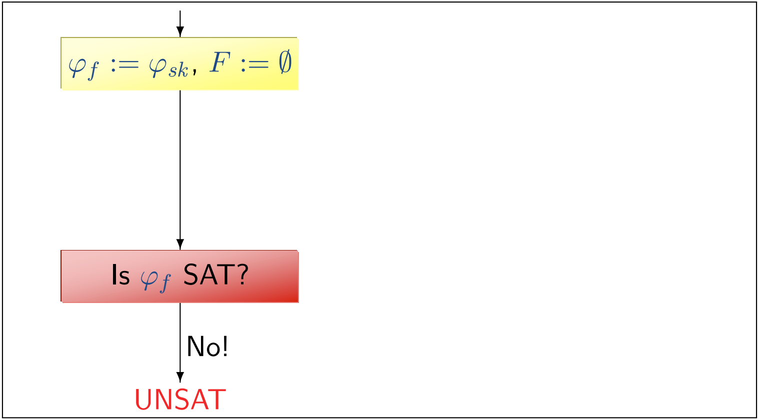
$\varphi_f := \varphi_{sk}$, $F := \emptyset$

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

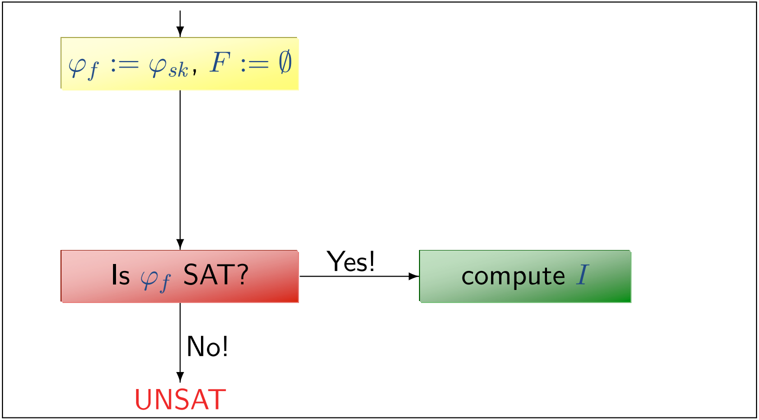
Incremental Flattening



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



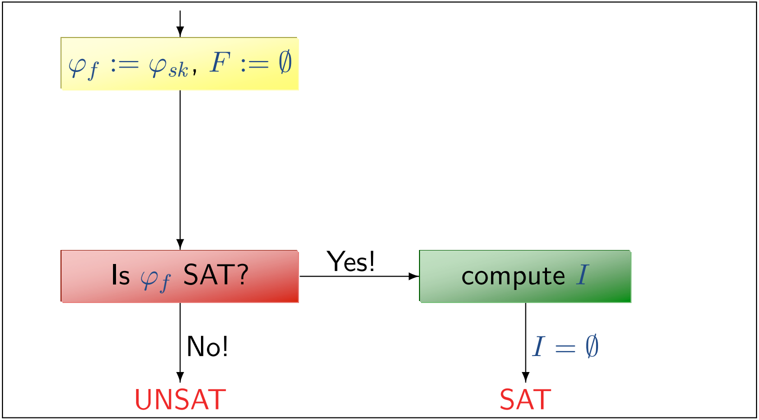
φ_{sk}: Boolean part of φ
F: set of terms that are in the encoding



φ_{sk}: Boolean part of φ

F: set of terms that are in the encoding

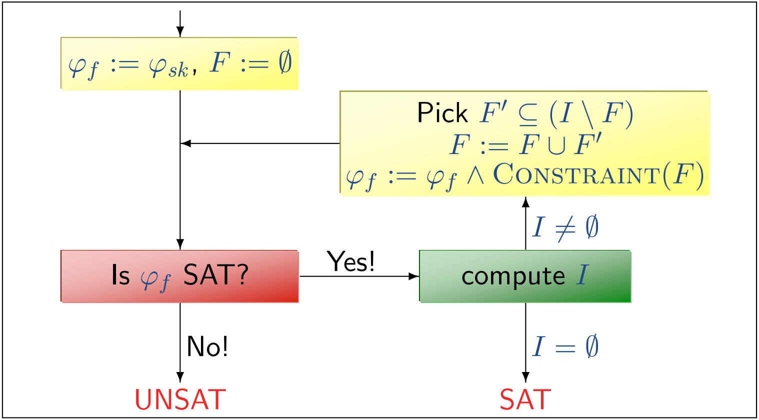
I: set of terms that are inconsistent with the current assignment



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



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

- 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.

Algorithm 6.3.1: INCREMENTAL-BV-FLATTENING

Input: A formula φ in bit-vector logic

Output: "Satisfiable" if the formula is satisfiable, and "Unsatisfiable" otherwise

```
function INCREMENTAL-BV-FLATTENING(φ)
    B := e(φ);                                   ▷ propositional skeleton of φ
    for each t_[l] ∈ T(φ) do
        for each i ∈ {0, ..., l − 1} do
            set e(t)_i to a new Boolean variable;
    while (TRUE) do
        α := SAT-SOLVER(B);
        if α="Unsatisfiable" then
            return "Unsatisfiable";
        else
            Let I ⊆ T(φ) be the set of terms that are inconsistent with the
                satisfying assignment;
            if I = ∅ then
                return "Satisfiable";
            else
                Select "easy" F′ ⊆ I;
                for each t_[l] ∈ F′ do
                    B:=B ∧ BV-CONSTRAINT(e, t);
```

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.

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