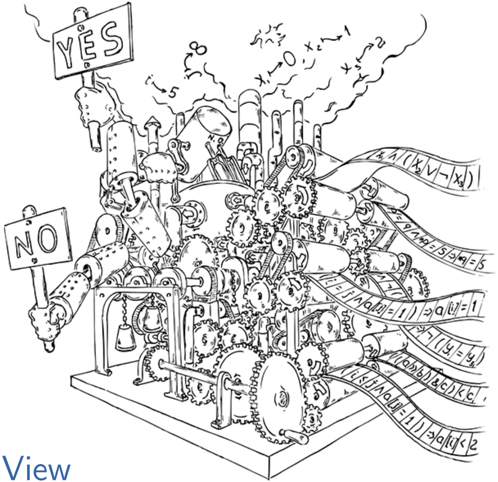
Bit-Vectors

Chapter 6



Decision Procedures

An Algorithmic Point of View

D.Kroening O.Strichman

Revision 1.2

Outline

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

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

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

- $\sim x$: bit-wise negation of x
- $ext(x)$: sign- or zero-extension of x
- $x << d$: 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

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

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

Examples

$$\begin{aligned}\langle 11001000\rangle_U &= 200\\ \langle 11001000\rangle_S &= -128+64+8=-56\\ \langle 01100100\rangle_S &= 100\end{aligned}$$

Notation to clarify width and encoding:

$$x_{[32]S}$$

Width and Encoding

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 $x(i)$.

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

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:

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

- A bit-wise OR:

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

$\Longrightarrow$ 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:

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

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

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

- 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!

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

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!

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

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

Semantics for addition, subtraction:

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

Semantics for Arithmetic Expressions

Semantics for addition, subtraction:

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

Semantics for addition, subtraction:

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

We can even mix the encodings:

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

Semantics for Relational Operators

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

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

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 $\mu(t)_i$.

Bit-vector Flattening

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

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

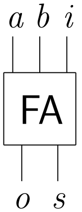
(read $x = y$ over bits as $x \iff y$)

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

How to flatten $a + b$?

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

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



Full Adder

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

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

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

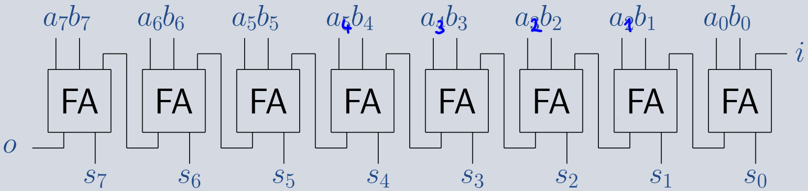
s: $(a \vee b \vee i \vee \neg s) \wedge (a \vee b \vee \neg i \vee s) \wedge (a \vee \neg b \vee i \vee s) \wedge$
$(a \vee b \vee \neg i \vee \neg s) \wedge (\neg a \vee b \vee i \vee s) \wedge (\neg a \vee b \vee \neg i \vee \neg s) \wedge$
$(\neg a \vee \neg b \vee i \vee \neg s) \wedge (\neg a \vee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

- Multipliers result in very hard formulas
- Example:

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

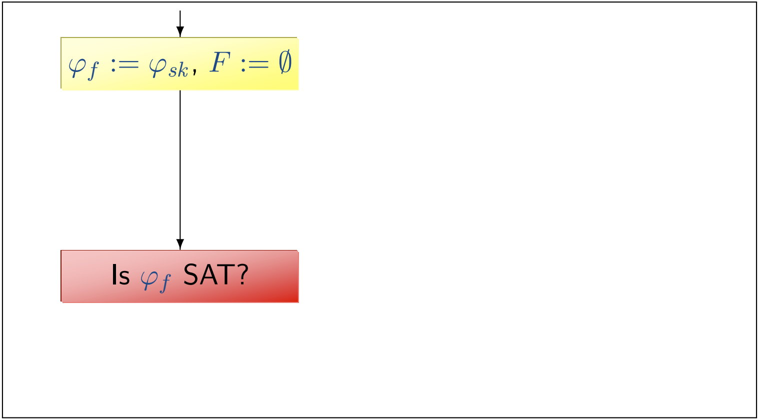
CNF: About 11000 variables, unsolvable for current SAT solvers

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

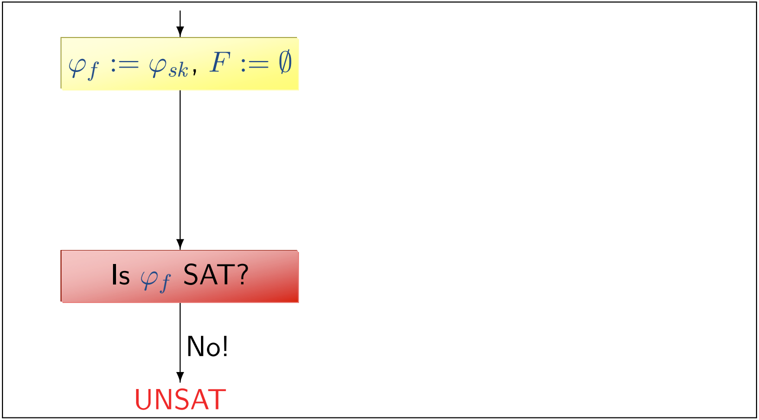
Incremental Flattening

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

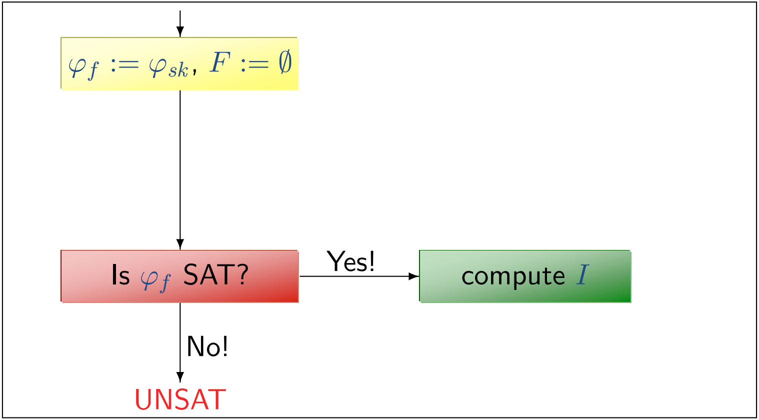
φ_{sk}: Boolean part of φ
F: set of terms that are in the encoding



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



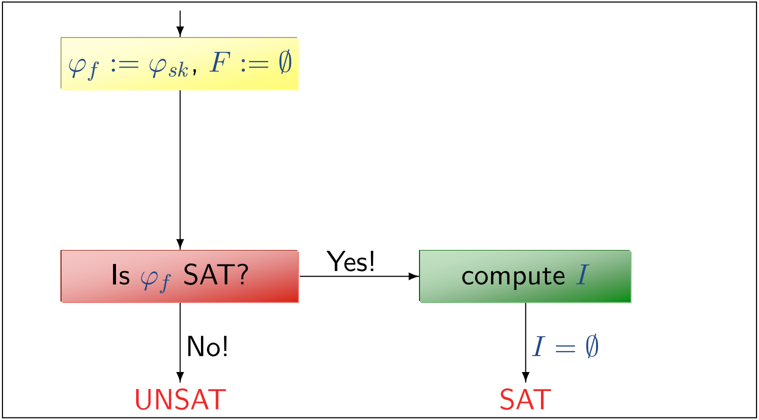
φ_{sk}: Boolean part of φ
F: set of terms that are in the encoding



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



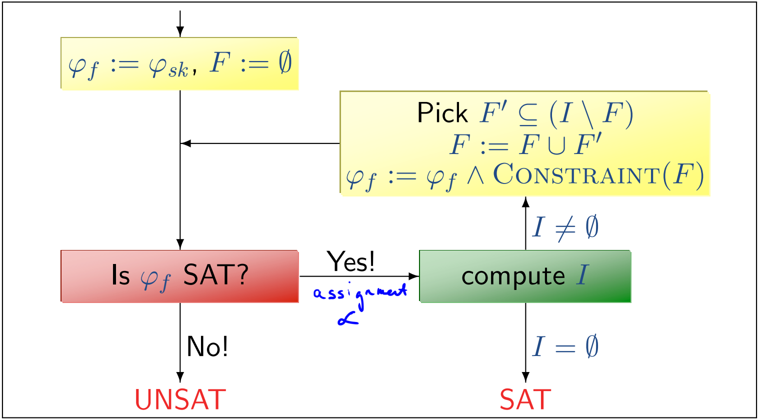
φ_{sk}: Boolean part of φ

F: set of terms that are in the encoding

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

Incremental Flattening

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?

- 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 ($a = b + 1$)