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";
    }
}

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
\[ \text{formula} : \ \text{formula} \lor \text{formula} \ | \ \neg \text{formula} \ | \ \text{atom} \\
\text{atom} : \ \text{term rel term} \ | \ \text{Boolean-Identifier} \ | \ \text{term[ constant]} \\
\text{rel} : = \ | < \\
\text{term} : \ \text{term op term} \ | \ \text{identifier} \ | \ \sim \text{term} \ | \ \text{constant} \ | \ \text{atom?term:term} \ | \\
\text{term[ constant : constant]} \ | \ \text{ext( term )} \\
\text{op} : + \ | - \ | . \ | / \ | \ll \ | \gg \ | \& \ | \lnot \ | \oplus \ | \circ \]
Bit-Vector Logic: Syntax

\[
\text{formula} : \quad \text{formula} \lor \text{formula} \mid \neg \text{formula} \mid \text{atom} \\
\text{atom} : \quad \text{term} \ \text{rel} \ \text{term} \mid \text{Boolean-Identifier} \mid \text{term} [\ \text{constant} ]
\]

\[
\text{rel} : \quad = \mid <
\]

\[
\text{term} : \quad \text{term} \ \text{op} \ \text{term} \mid \text{identifier} \mid \sim \ \text{term} \mid \text{constant} \mid \text{atom} \ ? \ \text{term} : \ \text{term}
\]

\[
\text{term} [\ \text{constant} : \ \text{constant} ] \mid \text{ext}(\ \text{term})
\]

\[
\text{op} : \quad + \mid - \mid \cdot \mid / \mid \ll \mid \gg \mid \& \mid \mid \mid \oplus \mid \circ
\]

- \sim x: bit-wise negation of \( x \)
- \text{ext}(x): sign- or zero-extension of \( x \)
- \( x \ll d \): left shift with distance \( d \)
- \( x \circ y \): concatenation of \( x \) and \( y \)
$(x - y > 0) \iff (x > y)$

Valid over $\mathbb{R}/\mathbb{N}$, but not over the bit-vectors.
(Many compilers have this sort of bug)
The meaning depends on the width and encoding of the variables.
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, . . .
\[ \langle 11001000 \rangle_U = 200 \]
\[ \langle 11001000 \rangle_S = -128 + 64 + 8 = -56 \]
\[ \langle 01100100 \rangle_S = 100 \]
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
Definition (Bit-Vector)

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

$$b : \{0, \ldots, l - 1\} \rightarrow \{0, 1\}$$
Definition (Bit-Vector)

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

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

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

We also write $x_i$ for $x(i)$. 
\[ \lambda \text{ expressions are functions without a name} \]
Lambda-Notation for Bit-Vectors

\[ \lambda \text{ expressions are functions without a name} \]

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:

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

- A bit-wise OR:

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

\[ \Rightarrow \text{we now have semantics for the bit-wise operators.} \]
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:

\[x_{[9]} = x_9\]
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:

\[x_{[9]} = x_9\]

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

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

```c
unsigned char number = 200;
number = number + 100;
printf("Sum: %d\n", number);
```
What is the output of the following program?

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

On most architectures, this is 44!

```
11001000  = 200
+ 01100100  = 100
= 00101100  = 44
```
What is the output of the following program?

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

On most architectures, this is 44!

```
11001000  = 200
+ 01100100  = 100
= 00101100  = 44
```

→ Bit-vector arithmetic uses modular arithmetic!
Semantics for addition, subtraction:

\[ a[l] +_U b[l] = c[l] \iff \langle a \rangle_U + \langle b \rangle_U = \langle c \rangle_U \pmod{2^l} \]

\[ a[l] -_U b[l] = c[l] \iff \langle a \rangle_U - \langle b \rangle_U = \langle c \rangle_U \pmod{2^l} \]
Semantics for addition, subtraction:

\[ a[l] +_U b[l] = c[l] \iff \langle a \rangle_U + \langle b \rangle_U = \langle c \rangle_U \mod 2^l \]
\[ a[l] -_U b[l] = c[l] \iff \langle a \rangle_U - \langle b \rangle_U = \langle c \rangle_U \mod 2^l \]
\[ a[l] +_S b[l] = c[l] \iff \langle a \rangle_S + \langle b \rangle_S = \langle c \rangle_S \mod 2^l \]
\[ a[l] -_S b[l] = c[l] \iff \langle a \rangle_S - \langle b \rangle_S = \langle c \rangle_S \mod 2^l \]
Semantics for addition, subtraction:

\[
\begin{align*}
a[l] +_U b[l] &= c[l] \iff \langle a \rangle_U + \langle b \rangle_U = \langle c \rangle_U \mod 2^l \\
a[l] -_U b[l] &= c[l] \iff \langle a \rangle_U - \langle b \rangle_U = \langle c \rangle_U \mod 2^l \\
a[l] +_S b[l] &= c[l] \iff \langle a \rangle_S + \langle b \rangle_S = \langle c \rangle_S \mod 2^l \\
a[l] -_S b[l] &= c[l] \iff \langle a \rangle_S - \langle b \rangle_S = \langle c \rangle_S \mod 2^l
\end{align*}
\]

We can even mix the encodings:

\[
\begin{align*}
a[l]U +_U b[l]S &= c[l]U \iff \langle a \rangle_U + \langle b \rangle_S = \langle c \rangle_U \mod 2^l
\end{align*}
\]
Semantics for relational operators:

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

$$a[U] < b[U] \iff \langle a \rangle_U < \langle b \rangle_U$$

$$a[S] < b[S] \iff \langle a \rangle_S < \langle b \rangle_S$$

Note that most compilers don't support comparisons with mixed encodings.
Semantics for Relational Operators

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

\[ a[U]_\text{L} < b[U]_\text{L} \iff \langle a \rangle_U < \langle b \rangle_U \]
\[ a[S]_\text{L} < b[S]_\text{L} \iff \langle a \rangle_S < \langle b \rangle_S \]

Mixed encodings:

\[ a[U]_\text{L} < b[S]_\text{L} \iff \langle a \rangle_U < \langle b \rangle_S \]
\[ a[S]_\text{L} < b[U]_\text{L} \iff \langle a \rangle_S < \langle b \rangle_U \]

Note that most compilers don’t support comparisons with mixed encodings.
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'*
A Simple Decision Procedure

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 $\mu(t)_i$. 
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 $a|_b b$:

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

(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 $a_i \| b_i$:

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

(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 $a + b$?
Flattening Bit-Vector Arithmetic

How to flatten $a + b$?

$\rightarrow$ we can build a circuit that adds them!

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FA</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The full adder:

- $s \equiv (a + b + i) \mod 2 \equiv a \oplus b \oplus i$
- $o \equiv (a + b + i) \div 2 \equiv a \cdot b + a \cdot i + b \cdot i$

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?
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:

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

CNF: About 11000 variables, unsolvable for current SAT solvers

Similar problems with division, modulo

Q: Why is this hard?
Q: How do we fix this?
\( \varphi_f := \varphi_{sk}, \ F := \emptyset \)

\( \varphi_{sk} \): Boolean part of \( \varphi \)

\( F \): set of terms that are in the encoding
Incremental Flattening

\( \varphi_f := \varphi_{sk}, F := \emptyset \)

Is \( \varphi_f \) SAT?

\( \varphi_{sk} \): Boolean part of \( \varphi \)

\( F \): set of terms that are in the encoding
\[ \varphi_f := \varphi_{sk}, \ F := \emptyset \]

Is \( \varphi_f \) SAT?

No!

UNSAT

\( \varphi_{sk} \): Boolean part of \( \varphi \)

\( F \): set of terms that are in the encoding
\[ \varphi_f := \varphi_{sk}, \ F := \emptyset \]

Is \( \varphi_f \) SAT?

Yes! \( \rightarrow \) compute \( I \)

No! \( \rightarrow \) UNSAT

\( \varphi_{sk} \): Boolean part of \( \varphi \)
\( F \): set of terms that are in the encoding
\( I \): set of terms that are inconsistent with the current assignment
Incremental Flattening

\[ \varphi_f := \varphi_{sk}, \quad F := \emptyset \]

Is \( \varphi_f \) SAT?

- Yes!
  - compute \( I \)
  - \( I = \emptyset \)
- No!
  - UNSAT

\( \varphi_{sk} \): Boolean part of \( \varphi \)

\( F \): set of terms that are in the encoding

\( I \): set of terms that are inconsistent with the current assignment
Incremental Flattening

\[ \varphi_f := \varphi_{sk}, \quad F := \emptyset \]

Pick \( F' \subseteq (I \setminus F) \)
\[ F := F \cup F' \]
\[ \varphi_f := \varphi_f \land \text{CONSTRAINT}(F) \]

Is \( \varphi_f \) SAT?
- Yes! compute \( I \)
- No! UNSAT
- \( I \neq \emptyset \)
- \( I = \emptyset \) SAT

\( \varphi_{sk} \): Boolean part of \( \varphi \)
\( F \): 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
- \( \varphi_f \) only gets stronger – use an **incremental SAT solver**
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 $\varphi_f$?

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 ($a = b + 1$)
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 $\varphi_f$?

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 $\varphi_f$?

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 ($a = b + 1$)