Decision Procedures and Verification

Martin Blicha

Charles University

23.4.2018

Theory of bit vector arithmetics

Bit vector arithmetics

Definition

A quantifier-free formula in the language of the theory of bit vector arithmetic is defined by the following grammar:

$$
\begin{aligned}
\mathit{fla} &: \mathit{fla} \wedge \mathit{fla} \mid \neg \mathit{fla} \mid \mathit{atom} \\
\mathit{atom} &: \mathit{term}\ \mathit{rel}\ \mathit{term} \mid \mathit{Boolean} - \mathit{Identifier} \mid \mathit{term}[\mathit{constant}] \\
\mathit{rel} &: = \mid < \\
\mathit{sum} &: \mathit{term} \mid \mathit{sum} + \mathit{term} \\
\mathit{term} &: \mathit{term}\ \mathit{op}\ \mathit{term} \mid \mathit{identifier} \mid \sim\mathit{term} \mid \mathit{constant} \mid \\
&\quad \mathit{atom}?\mathit{term} : \mathit{term} \mid \mathit{term}[\mathit{constant} : \mathit{constant}] \mid \mathit{ext}(\mathit{term}) \\
\mathit{op} &: + \mid - \mid \cdot \mid / \mid \ll \mid \gg \mid \& \mid | \mid \oplus \mid \circ
\end{aligned}
$$

Motivation (1)

- Consider a bit vector arithmetic formula φ:

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

  - Valid over integers
  - Not valid in structure with bit-vectors of *fixed* length

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

- The meaning of arithmetic operations is defined by means of *modular* arithmetic.

Motivation (2)

- Efficient programming on bit-level
  - Encoding literals in SAT solver

```
unsigned variable_index
    (int lit){
if(lit < 0)
  return -lit;
return lit;
}
```

```
unsigned variable_index
    (unsigned lit){
return lit >> 1;
}
```

```
bool sign(unsigned lit)
    {
return lit & 1;
}
```

Notation

- **Church's λ-notation** will be used to define bit-vectors
- λ-expression for a bit vector of length l:
  - $\lambda i \in \{0, 1, \ldots, l-1\}.f(i)$, where $f(i)$ is an expression determining the value of the i-th bit
- Examples:
  - $\lambda i \in \{0, 1, \ldots, l-1\}.0$ is a bit vector of length l consisting of all 0
  - $\lambda i \in \{0, 1, \ldots, 7\}. \begin{cases} 0 \text{ if I is even} \\ 1 \text{ otherwise} \end{cases}$ is a bit vector 10101010
  - $\lambda i \in \{0, 1, \ldots, l-1\}.\neg b_i$ is a bit vector of length l corresponding to bit-wise negation of a bit vector b

Semantics of operators (1)

Definition

Bit vector b of length l is an assignment $b : \{0, 1, \ldots, l-1\} \to \{0, 1\}$. The i-th bit of bit vector b is denoted as b_i. The set of all the bit vectors of length l is denoted as $bvec_l$.

- The length of the bit-vectors has impact on the satisfiability of a formulas.
- *Signed* and *unsigned* bit vectors are distinguished.
  - semantics of arithmetic operations reflects the sign
  - The type of an expression is a pair:
    - the *width* in bits
    - whether is it signed or unsigned

Semantics of operators (2)

- Bit-wise negation $\sim$:
  - $\sim_{[l]}$: $bvec_l \rightarrow bvec_l$, where $\sim_{[l]} b = \lambda i.\neg b_i$
- Bit-wise and &:
  - $\&_{[l]} : bvec_l \times bvec_l \rightarrow bvec_l$, where $a \&_{[l]} b = \lambda i. a_i \wedge b_i$
- Bit-wise or $|$:
  - $|_{[l]}$: $bvec_l \times bvec_l \rightarrow bvec_l$, where $a \,|_{[l]}\, b = \lambda i. a_i \vee b_i$
- Bit-wise xor $\oplus$:
  - $\oplus_{[l]} : bvec_l \times bvec_l \rightarrow bvec_l$, where $a \oplus_{[l]} b = \lambda i. a_i \oplus b_i$
- Concatenation of bit-vectors $\circ$:
  - $\circ_{[l+k]} : bvec_l \times bvec_k \rightarrow bvec_{l+k}$, where

    $$a \circ_{[l+k]} b = \lambda i. \begin{cases} a_i : i < l \\ b_{i-l} : \text{ otherwise} \end{cases}$$

Semantics of operators (3)

- Encoding of natural numbers (unsigned):

Definition (binary encoding)

Let x be a natural number and $b \in bvec_l$ a bit vector. We say that b is a binary encoding of x if and only if: $x = <b>_U$, where $<>_U : bvec_l \rightarrow \{0, 1, \ldots, 2^l - 1\}$ and $<b>_U = \sum_{i=0}^{l-1} b_i 2^i$. Bit b_0 is the lowest bit, bit b_{l-1} is the highest bit.

- Encoding of natural integers (signed):

Definition (two's complement)

Let x be an integer and $b \in bvec_l$ a bit vector. $x = <b>_S$, where $<>_S : bvec_l \rightarrow \{-2^{l-1}, \ldots, 2^{l-1} - 1\}$ and $<b>_S = -2^{l-1} b_{l-1} + \sum_{i=0}^{l-2} b_i 2^i$. Bit $b_l - 1$ is called the *sign* bit of b.

Semantics of operators (4)

- addition and subtraction
  - $a_{[l]} +_U b_{[l]} = c_{[l]} \Leftrightarrow <a>_U + <b>_U = <c>_U \mod 2^l$
  - $a_{[l]} -_U b_{[l]} = c_{[l]} \Leftrightarrow <a>_U - <b>_U = <c>_U \mod 2^l$
  - $a_{[l]} +_S b_{[l]} = c_{[l]} \Leftrightarrow <a>_S + <b>_S = <c>_S \mod 2^l$
  - $a_{[l]} -_S b_{[l]} = c_{[l]} \Leftrightarrow <a>_S - <b>_S = <c>_S \mod 2^l$
- operations can be defined over mixed types
  - $a_{[l]U} +_U b_{[l]S} = c_{[l]U} \Leftrightarrow <a>_U + <b>_S = <c>_U \mod 2^l$
- unary minus
  - $-a_l = b_l \Leftrightarrow -<a>_S = <b>_S \mod 2^l$

Semantics of operators (5)

- multiplication and division
  - $a_{[l]} *_U b_{[l]} = c_{[l]} \Leftrightarrow <a>_U * <b>_U = <c>_U \bmod 2^l$
  - $a_{[l]} /_U b_{[l]} = c_{[l]} \Leftrightarrow <a>_U / <b>_U = <c>_U \bmod 2^l$
  - $a_{[l]} *_S b_{[l]} = c_{[l]} \Leftrightarrow <a>_S * <b>_S = <c>_S \bmod 2^l$
  - $a_{[l]} /_S b_{[l]} = c_{[l]} \Leftrightarrow <a>_S / <b>_S = <c>_S \bmod 2^l$
- relation operators
  - $a_{[l]U} < b_{[l]U} \Leftrightarrow <a>_U < <b>_U$
  - $a_{[l]S} < b_{[l]S} \Leftrightarrow <a>_S < <b>_S$
  - $a_{[l]U} < b_{[l]S} \Leftrightarrow <a>_U < <b>_S$
  - $a_{[l]S} < b_{[l]U} \Leftrightarrow <a>_S < <b>_U$

Semantics of operators (6)

- extension of a bit vector *ext*
  - bit vector of length l is extended to length m for $l \leq m$:
    - *zero extension*: $ext_{[m]U}(a_{[l]}) = b_{[m]U} \Leftrightarrow <a>_U = <b>_U$
    - *sign extension*: $ext_{[m]S}(a_{[l]}) = b_{[m]S} \Leftrightarrow <a>_S = <b>_S$
- shifting of a bit vector
  - left shift - zero bits are filled from rigth
    - $a_{[l]} \ll b_U = \lambda i. \begin{cases} a_{i-<b>} \text{ if } i \geq <b>_U \\ 0 : \text{otherwise} \end{cases}$
- right shift - distinguished operations for signed and unsigned case:
  - $a_{[l]U} \gg b_U = \lambda i. \begin{cases} a_{i+<b>} \text{ if } i < l - <b>_U \\ 0 : \text{otherwise} \end{cases}$
  - $a_{[l]S} \gg b_U = \lambda i. \begin{cases} a_{i+<b>} \text{ if } i < l - <b>_U \\ a_{l-1} : \text{otherwise} \end{cases}$

Bit-vector flattening

- For a given bit-vector formula φ and equisatisfiable propositional ψ is constructed.

---

procedure BV-FLATTENING(φ)
    $\mathcal{B} \leftarrow e(\varphi)$
    for each $t_{[l]} \in T(\varphi)$ **do**
        for $i \in 0, 1, \ldots, l-1$ **do**
            set $e(t)_i$ to a new Boolean variable
    for each $a \in At(\varphi)$ **do**
        $\mathcal{B} \leftarrow \mathcal{B} \wedge$ BV-CONSTRAINT(e, a)
    for each $t_{[l]} \in T(\varphi)$ **do**
        $\mathcal{B} \leftarrow \mathcal{B} \wedge$ BV-CONSTRAINT(e, t)

- e is a propositional encoder, $At(\varphi)$ and $T(\varphi)$ a set of atoms and terms of φ, respectively.

Bit vector constraints (1)

- If t is a bit vector or a is a propositional variable, no constraint is needed.
  - $\text{BV-CONSTRAINT}(e, t)$ and $\text{BV-CONSTRAINT}(e, a)$ return True.
- If t is a vector constant $C_{[l]}$ then
  - $\text{BV-CONSTRAINT}(e, t)$ returns $\bigwedge_{i=0}^{l-1}(C_i \Leftrightarrow e(t)_i)$
- If t contains bit-wise operator then
  - if $t = \sim_{[l]} a$ $\text{BV-CONSTRAINT}(e, t)$ returns $\bigwedge_{i=0}^{l-1}(\neg a_i \Leftrightarrow e(t)_i)$
  - if $t = a \&_{[l]} b$ $\text{BV-CONSTRAINT}(e, t)$ returns $\bigwedge_{i=0}^{l-1}(a_i \wedge b_i \Leftrightarrow e(t)_i)$
  - if $t = a \mid_{[l]} b$ $\text{BV-CONSTRAINT}(e, t)$ returns $\bigwedge_{i=0}^{l-1}(a_i \vee b_i \Leftrightarrow e(t)_i)$
  - if $t = a \oplus_{[l]} b$ $\text{BV-CONSTRAINT}(e, t)$ returns $\bigwedge_{i=0}^{l-1}(a_i \oplus b_i \Leftrightarrow e(t)_i)$
  - if $t = a_{[l]} \circ_{[l+k]} b_{[k]}$ $\text{BV-CONSTRAINT}(e, t)$ returns
    $$\bigwedge_{i=0}^{l+k-1} \begin{cases} (a_i \Leftrightarrow e(t)_i) : & \text{if } i < l \\ (b_i \Leftrightarrow e(t)_i) : & \text{otherwise} \end{cases}$$

Bit vector constraints (2)

- Constraints for arithmetic operations are based on implementations of these operations in logic circuits
  - Various implementations
  - Simplest usually burden the SAT solver the least
- A *full adder* is defined using the two functions *carry* and *sum*. Both of these functions take three input bits a, b, and cin as arguments. The function carry calculates the carry-out bit of the adder, and the function sum calculates the sum bit:
  - $carry(a, b, cin) = (a \wedge b) \vee ((a \oplus b) \wedge cin)$
  - $sum(a, b, cin) = (a \oplus b) \oplus cin$
- *Carry* bits $c_0, c_1, \ldots, c_l$ for l-bit vectors x and y with *cin* the input carry bits are defined as
  - $c_i = \begin{cases} cin \text{ if } i = 0 \\ carry(x_{i-1}, y_{i-1}, c_{i-1}) \text{ otherwise} \end{cases}$

Bit vector constraints (3)

- *l-bit adder*: A funtion add that assigns two l-bit bit vectors x and y and input carry bit cin an l-bit bit vector r corresponding to their sum and a carry-out bit cout is called l-bit added. The function add is defined as follows:
  - $add(x, y, cin) = (r, cout)$
  - $r_i = sum(x_i, y_i, c_i)$ for $i = 0, \ldots, l-1$
  - $cout = c_l$, where c_i for $i = 0, \ldots, l$ are carry bits
- Constraint $t = a +_{[l]} b$ can be encoded by l-bit adder where the input carry bit is 0:
  - $\text{BV-C}\textsc{onstraint}(e, t)$ returns $\bigwedge_{i=0}^{l-1} (add(a, b, 0).r_i \Leftrightarrow e(t)_i)$.
    - Because $\langle a \rangle_U + \langle b \rangle_U = \langle e(t) \rangle_U \mod 2^l$ iff $\bigwedge_{i=0}^{l-1} (add(a, b, 0).r_i \Leftrightarrow e(t)_i)$.
- Constraint $t = a -_{[l]} b$ can be encoded in a similar way:
  - $\text{BV-C}\textsc{onstraint}(e, t)$ returns $\bigwedge_{i=0}^{l-1} (add(a, \sim b, 1).r_i \Leftrightarrow e(t)_i)$
    - Uses the fact that $\langle (\sim b + 1) \rangle_S = -\langle b \rangle_S \mod 2^l$.

Bit vector constraints (4)

- Relation operator constraints
  - For $at =_{def} (a =_{[l]} b)$ BV-CONSTRAINT(e, at) returns
    $(\bigwedge_{i=0}^{l-1}(a_i = b_i)) \Leftrightarrow e(at)$.
  - $a < b$ is transformed to $a - b < 0$ and adder is built for the subtraction. The result depends on the encoding.
    - Signed case: BV-CONSTRAINT(e, at) returns $\neg add(a, \sim b, 1).cout$
    - Unsigned case: BV-CONSTRAINT(e, at) returns $a_{l-1} \Leftrightarrow b_{l-1} \oplus add(a, b, 1).cout$
- Bit-vector shifting constraints
  - Assumptions: Shifted vector has l bits where l is a power of 2, size of the shift uses $n = log_2 l$ bits.
  - *Barrel* shifter is used.
    - Operates in n phases.
    - Stage s can shift the operand by 2^s bits or leave it unaltered.

Bit vector constraints (5)

- Barrel shifter constraints
  - For $t = a_{[l]} \ll b_{[n]}$ a function *lsh* for $s \in \{-1, 0, \ldots, n-1\}$ is defined as follows:
  - $lsh(a, b, -1) = a$
  - $lsh(a, b, s) = \lambda i \in \{0, \ldots, l-1\}. \begin{cases} (lsh(a, b, s-1))_{i-2^s} & \text{if } i \geq 2^s \wedge b_s \\ (lsh(a, b, s-1))_i & \text{if } \neg b_s \\ 0 & \text{otherwise} \end{cases}$
  - $\text{BV-CONSTRAINT}(e, t)$ returns $\bigwedge_{i=0}^{l-1} ((lsh(a, b, n)_i \Leftrightarrow e(t)_i)$.
- Multiplication constraints
  - For $t = a * b$ addition and shifts will be used, a function *mul* for $s \in \{-1, 0, \ldots, l-1\}$ is defined as follows:
    - $mul(a, b, -1) = 0$
    - $mul(a, b, s) = mul(a, b, s-1) + (b_s?(a \ll s) : 0)$
  - $\text{BV-CONSTRAINT}(e, t)$ returns $\bigwedge_{i=0}^{l-1} ((mul(a, b, l)_i \Leftrightarrow e(t)_i)$.

Bit vector constraints (6)

- Division constraints
  - For $t = a /_{[U]} b$ following constraints will be used:
    - $b \neq 0 \Rightarrow e(t) \cdot b + r = a$
    - $b \neq 0 \Rightarrow r < b$
  - Both constraints are returned by $\text{BV-C}\textsc{onstraint}(e, t)$ and r is a new bit vector the same width as b representing the remainder
  - Signed division and modulo operations are handled similarly.
- Conditional expression
  - Let $t = at?t_1 : t_2$ be a conditional expression where at is an atom and t_1, t_2 are terms.
  - $\text{BV-C}\textsc{onstraint}(e, t)$ returns

$$(at \Rightarrow \bigwedge_{i=0}^{l-1} (e(t)_i \Leftrightarrow e(t_1)_i)) \wedge (\neg at \Rightarrow \bigwedge_{i=0}^{l-1} (e(t)_i \Leftrightarrow e(t_2)_i))$$

Problems

- Constraints generated can be very long and complicated
  - Especially for 64-bits representation.
  - Multiplication of two n-bit numbers:
    - n=16 $\Rightarrow$ 1265 variables and 4177 clauses.
    - n=32 $\Rightarrow$ 5089 variables and 17057 clauses.
    - n=64 $\Rightarrow$ 20417 variables and 68929 clauses.
- Heuristics in SAT solvers are biased towards variables appearing frequently
  - $\varphi =_{def} (a \cdot b = c) \wedge (a \cdot b \neq c) \wedge (x < y) \wedge (x > y)$
  - SAT solver can focus on first part, ignoring the second part, which is much easier.

Incremental bit-flattening

- Idea: add constraints gradually
- Start with propositional skeleton, check satisfiability
  - UNSAT $\Rightarrow$ original formula is UNSAT
  - SAT $\Rightarrow$ add constraints that are violated by the satisfying assignment.
- Repeat until UNSAT or no constraints are violated by satisfying assignment.
- Incremental bit-flattening can be combined with uninterpreted functions to preserve functional consistency without adding constraints for particular operator

Incremental bit-flattening

procedure INCREMENTAL-BV-FLATTENING(φ)
    $\mathcal{B} \leftarrow e(\varphi)$
    for each $t_{[l]} \in T(\varphi)$ **do**
        for $i \in 0, 1, \ldots, l-1$ **do**
            set $e(t)_i$ to a new Boolean variable
    while TRUE **do**
        $\alpha \leftarrow$ SAT-SOLVER($\mathcal{B}$)
        if $\alpha = \mathit{UNSAT}$ **then return** UNSAT
        Let $I \subseteq T(\varphi)$ be the set of terms inconsistent with α
        if $I = \emptyset$ **then return** SAT
        Select $F \subseteq I$
        for each $t_{[l]} \in F$ **do** $\mathcal{B} \leftarrow \mathcal{B} \wedge$ BV-CONSTRAINT(e, t)