Satisfiability Modulo Theories
Lezione 2 - An Eager Approach: Solving Bit-Vectors

(slides revision: Saturday 14th March, 2015, 11:46)

Roberto Bruttomesso

Seminario di Logica Matematica
(Corso Prof. Silvio Ghilardi)

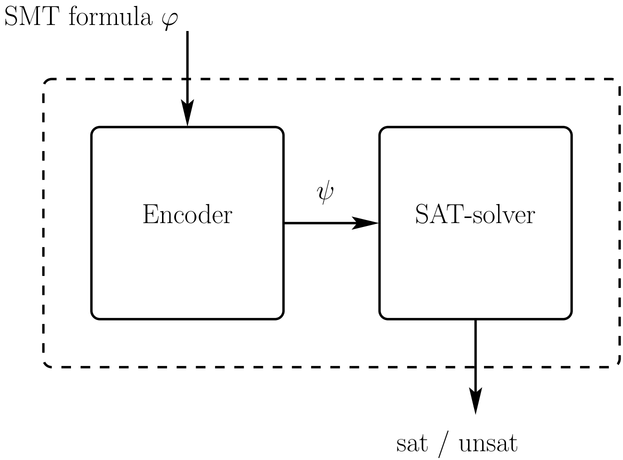
27 Ottobre 2011

Recall from last lecture ...

Approaches to solve SMT formulæ are based on the observation that SMT can be **reduced** to SAT, i.e., the purely Boolean Satisfiability Problem

Recall from last lecture . . .

Approaches to solve SMT formulæ are based on the observation that SMT can be **reduced** to SAT, i.e., the purely Boolean Satisfiability Problem



Outline

1. Bit-Vectors
   - Syntax
   - Semantic
   - Examples

2. Solving Bit-Vectors
   - Bit-Blasting
   - Simplifications

Introduction

Bit-Vectors are extremely useful data structures, used to symbolically represent hardware and software constructs (see later)

Introduction

Bit-Vectors are extremely useful data structures, used to symbolically represent hardware and software constructs (see later)

The world of Bit-Vectors is a **finite** world, i.e., with Bit-Vectors it is not possible to represent/handle arbitrarily large numbers

Introduction

Bit-Vectors are extremely useful data structures, used to symbolically represent hardware and software constructs (see later)

The world of Bit-Vectors is a **finite** world, i.e., with Bit-Vectors it is not possible to represent/handle arbitrarily large numbers

Indeed, when speaking about Bit-Vectors we always associate a **width** (which is usually a power of 2, often 32 or 64)

Introduction

Bit-Vectors are extremely useful data structures, used to symbolically represent hardware and software constructs (see later)

The world of Bit-Vectors is a **finite** world, i.e., with Bit-Vectors it is not possible to represent/handle arbitrarily large numbers

Indeed, when speaking about Bit-Vectors we always associate a **width** (which is usually a power of 2, often 32 or 64)

The width specifies the (maximum) **number of bits** used to represent variables and terms

Introduction

Bit-Vectors are extremely useful data structures, used to symbolically represent hardware and software constructs (see later)

The world of Bit-Vectors is a **finite** world, i.e., with Bit-Vectors it is not possible to represent/handle arbitrarily large numbers

Indeed, when speaking about Bit-Vectors we always associate a **width** (which is usually a power of 2, often 32 or 64)

The width specifies the (maximum) **number of bits** used to represent variables and terms

Bit-Vector formulæ are mathematically characterized by the theory of Bit-Vectors $\mathcal{BV}$

Bit-Vectors

A bit-vector is an array of bits

2	1	0
1	1	0

$a_{[3]}$

2	1	0
0	1	1

$b_{[3]}$

Bit-Vectors

A bit-vector is an array of bits

2	1	0
1	1	0

$a_{[3]}$

2	1	0
0	1	1

$b_{[3]}$

Selection (or Extraction): $a_{[3]}[1:0]$

1	0
1	0

Bit-Vectors

A bit-vector is an array of bits

2	1	0
1	1	0

$a_{[3]}$

2	1	0
0	1	1

$b_{[3]}$

Selection (or Extraction): $a_{[3]}[1:0]$

1	0
1	0

Notice that

- $a_{[n]}[i:j]$ returns a Bit-Vector of width $i - j + 1$ $(0 \leq j \leq i \leq n-1)$

Bit-Vectors

A bit-vector is an array of bits

$$\underset{a_{[3]}}{\begin{array}{ccc} \scriptstyle 2 & \scriptstyle 1 & \scriptstyle 0 \\ \hline \multicolumn{1}{|c|}{1} & \multicolumn{1}{c|}{1} & \multicolumn{1}{c|}{0} \\ \hline \end{array}} \qquad \underset{b_{[3]}}{\begin{array}{ccc} \scriptstyle 2 & \scriptstyle 1 & \scriptstyle 0 \\ \hline \multicolumn{1}{|c|}{0} & \multicolumn{1}{c|}{1} & \multicolumn{1}{c|}{1} \\ \hline \end{array}}$$

Selection (or Extraction): $a_{[3]}[1:0]$

$$\begin{array}{cc} \scriptstyle 1 & \scriptstyle 0 \\ \hline \multicolumn{1}{|c|}{1} & \multicolumn{1}{c|}{0} \\ \hline \end{array}$$

Notice that

- $a_{[n]}[i:j]$ returns a Bit-Vector of width $i - j + 1$ $(0 \leq j \leq i \leq n-1)$
- $a_{[n]}[n-1:0]$

Bit-Vectors

A bit-vector is an array of bits

2	1	0
1	1	0

$a_{[3]}$

2	1	0
0	1	1

$b_{[3]}$

Selection (or Extraction): $a_{[3]}[1:0]$

1	0
1	0

Notice that

- $a_{[n]}[i:j]$ returns a Bit-Vector of width $i - j + 1$ $(0 \leq j \leq i \leq n-1)$
- $a_{[n]}[n-1:0] = a_{[n]}$

Bit-Vectors

A bit-vector is an array of bits

2	1	0
1	1	0

$a_{[3]}$

2	1	0
0	1	1

$b_{[3]}$

Selection (or Extraction): $a_{[3]}[1:0]$

1	0
1	0

Notice that

- $a_{[n]}[i:j]$ returns a Bit-Vector of width $i-j+1$ $(0 \leq j \leq i \leq n-1)$
- $a_{[n]}[n-1:0] = a_{[n]}$
- Selection has precedence over any other operator

Bit-Vectors

A bit-vector is an array of bits

2	1	0
1	1	0

$a_{[3]}$

2	1	0
0	1	1

$b_{[3]}$

Concatenation $a_{[3]} :: b_{[3]}$

5	4	3	2	1	0
1	1	0	0	1	1

Bit-Vectors

A bit-vector is an array of bits

2	1	0
1	1	0

$a_{[3]}$

2	1	0
0	1	1

$b_{[3]}$

Concatenation $a_{[3]} :: b_{[3]}$

5	4	3	2	1	0
1	1	0	0	1	1

Notice that

- $a_{[n]} :: b_{[m]}$ returns a Bit-Vector of width $n + m$

Bit-Vectors

A bit-vector is an array of bits

2	1	0
1	1	0

$a_{[3]}$

2	1	0
0	1	1

$b_{[3]}$

Concatenation $a_{[3]} :: b_{[3]}$

5	4	3	2	1	0
1	1	0	0	1	1

Notice that

- $a_{[n]} :: b_{[m]}$ returns a Bit-Vector of width $n + m$
- $a_{[n]}[n-1:i] :: a_{[n]}[i-1:0]$

A bit-vector is an array of bits

2	1	0
1	1	0

$a_{[3]}$

2	1	0
0	1	1

$b_{[3]}$

Concatenation $a_{[3]} :: b_{[3]}$

5	4	3	2	1	0
1	1	0	0	1	1

Notice that

- $a_{[n]} :: b_{[m]}$ returns a Bit-Vector of width $n + m$
- $a_{[n]}[n-1:i] :: a_{[n]}[i-1:0] = a_{[n]}[n-1:0]$

Bit-Vectors

A bit-vector is an array of bits

2	1	0
1	1	0

$a_{[3]}$

2	1	0
0	1	1

$b_{[3]}$

Concatenation $a_{[3]} :: b_{[3]}$

5	4	3	2	1	0
1	1	0	0	1	1

Notice that

- $a_{[n]} :: b_{[m]}$ returns a Bit-Vector of width $n + m$
- $a_{[n]}[n-1:i] :: a_{[n]}[i-1:0] = a_{[n]}[n-1:0] = a_{[n]}$

Bit-Vectors

A bit-vector is an array of bits

2	1	0
1	1	0

$a_{[3]}$

2	1	0
0	1	1

$b_{[3]}$

Arithmetic $a_{[3]} + b_{[3]}$

	2	1	0
1	0	0	1

Bit-Vectors

A bit-vector is an array of bits

2	1	0
1	1	0

$a_{[3]}$

2	1	0
0	1	1

$b_{[3]}$

Arithmetic $a_{[3]} + b_{[3]}$

	2	1	0
1	0	0	1

Notice that

- To be precise, we should have written $a_{[3]} +_{[3]} b_{[3]}$ (widths must be the same)
- Semantic is that of **modular** arithmetic

Bit-Vectors

A bit-vector is an array of bits

2	1	0
1	1	0

$a_{[3]}$

2	1	0
0	1	1

$b_{[3]}$

Bitwise $a_{[3]}$ **AND** $b_{[3]}$

2	1	0
0	1	0

Bit-Vectors

A bit-vector is an array of bits

2	1	0
1	1	0

$a_{[3]}$

2	1	0
0	1	1

$b_{[3]}$

Bitwise $a_{[3]}$ **AND** $b_{[3]}$

2	1	0
0	1	0

Notice that

- Again, to be precise, we should have written $a_{[3]}$ **AND** $_{[3]} b_{[3]}$ (widths must be the same)
- Used to compute bit-mask operations

A (non-exhaustive) list of operators and predicates

Each Bit-Vector term of width n, is associated with a sort $\texttt{BV}_{[n]}$ $(n \geq 1)$

A (non-exhaustive) list of operators and predicates

Each Bit-Vector term of width n, is associated with a sort $\mathtt{BV}_{[n]}$ $(n \geq 1)$

Name	Symb	Type	Signature
Selection	$\_[i:j]$	Core	$\mathtt{BV}_{[n]} \to \mathtt{BV}_{[i-j+1]}$
Concatenation	$::$		$\mathtt{BV}_{[n]} \times \mathtt{BV}_{[m]} \to \mathtt{BV}_{[n+m]}$
Addition	$+$	Arith.	$\mathtt{BV}_{[n]} \times \mathtt{BV}_{[n]} \to \mathtt{BV}_{[n]}$
Subtraction	$-$		$\mathtt{BV}_{[n]} \times \mathtt{BV}_{[n]} \to \mathtt{BV}_{[n]}$
Multiplication	$*$		$\mathtt{BV}_{[n]} \times \mathtt{BV}_{[n]} \to \mathtt{BV}_{[n]}$
Less than (signed)	$<_s$		$\mathtt{BV}_{[n]} \times \mathtt{BV}_{[n]} \to \mathtt{Bool}$
Less than (unsigned)	$<_u$		$\mathtt{BV}_{[n]} \times \mathtt{BV}_{[n]} \to \mathtt{Bool}$
Bitwise and	**AND**	Bitwise	$\mathtt{BV}_{[n]} \times \mathtt{BV}_{[n]} \to \mathtt{BV}_{[n]}$
Bitwise or	**OR**		$\mathtt{BV}_{[n]} \times \mathtt{BV}_{[n]} \to \mathtt{BV}_{[n]}$
Bitwise not	**NOT** _		$\mathtt{BV}_{[n]} \to \mathtt{BV}_{[n]}$

A (non-exhaustive) list of operators and predicates

Each Bit-Vector term of width n, is associated with a sort $\mathtt{BV}_{[n]}$ $(n \geq 1)$

Name	Symb	Type	Signature
Selection	$\_[i:j]$	Core	$\mathtt{BV}_{[n]} \to \mathtt{BV}_{[i-j+1]}$
Concatenation	$::$		$\mathtt{BV}_{[n]} \times \mathtt{BV}_{[m]} \to \mathtt{BV}_{[n+m]}$
Addition	$+$	Arith.	$\mathtt{BV}_{[n]} \times \mathtt{BV}_{[n]} \to \mathtt{BV}_{[n]}$
Subtraction	$-$		$\mathtt{BV}_{[n]} \times \mathtt{BV}_{[n]} \to \mathtt{BV}_{[n]}$
Multiplication	$*$		$\mathtt{BV}_{[n]} \times \mathtt{BV}_{[n]} \to \mathtt{BV}_{[n]}$
Less than (signed)	$<_s$		$\mathtt{BV}_{[n]} \times \mathtt{BV}_{[n]} \to \mathtt{Bool}$
Less than (unsigned)	$<_u$		$\mathtt{BV}_{[n]} \times \mathtt{BV}_{[n]} \to \mathtt{Bool}$
Bitwise and	**AND**	Bitwise	$\mathtt{BV}_{[n]} \times \mathtt{BV}_{[n]} \to \mathtt{BV}_{[n]}$
Bitwise or	**OR**		$\mathtt{BV}_{[n]} \times \mathtt{BV}_{[n]} \to \mathtt{BV}_{[n]}$
Bitwise not	**NOT** $\_$		$\mathtt{BV}_{[n]} \to \mathtt{BV}_{[n]}$

Moreover, we have constants, e.g., $101101_{[6]}$

Bit-Vector semantic

Each sort $\mathtt{BV}_{[n]}$ is associated with a domain $D_n = \{0, 1, \ldots, 2^{n-1}\}$

Bit-Vector semantic

Each sort $\mathtt{BV}_{[n]}$ is associated with a domain $D_n = \{0, 1, \ldots, 2^{n-1}\}$
For example $\mathtt{BV}_{[4]}$ is associated with $D_4 = \{0, 1, \ldots, 15\}$

Bit-Vector semantic

Each sort $\texttt{BV}_{[n]}$ is associated with a domain $D_n = \{0, 1, \ldots, 2^{n-1}\}$
For example $\texttt{BV}_{[4]}$ is associated with $D_4 = \{0, 1, \ldots, 15\}$

As usual, the semantic for the other terms depends on a particular **assignment** to the variables

Bit-Vector semantic

Each sort $\mathtt{BV}_{[n]}$ is associated with a domain $D_n = \{0, 1, \ldots, 2^{n-1}\}$
For example $\mathtt{BV}_{[4]}$ is associated with $D_4 = \{0, 1, \ldots, 15\}$

As usual, the semantic for the other terms depends on a particular **assignment** to the variables

Each variable $x_{[n]}$ is associated with a function $[\![x_{[n]}]\!]$ of type $D_n \rightarrow \{0, 1\}$

Bit-Vector semantic

Each sort $\texttt{BV}_{[n]}$ is associated with a domain $D_n = \{0, 1, \ldots, 2^{n-1}\}$
For example $\texttt{BV}_{[4]}$ is associated with $D_4 = \{0, 1, \ldots, 15\}$

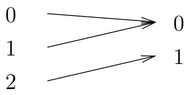
As usual, the semantic for the other terms depends on a particular **assignment** to the variables

Each variable $x_{[n]}$ is associated with a function $[\![x_{[n]}]\!]$ of type $D_n \to \{0, 1\}$

$[\![x_{[3]}]\!]$:

1	0	0

$[\![x_{[3]}]\!]$:



$nat_3([\![x_{[3]}]\!])$: 4

Bit-Vector semantic

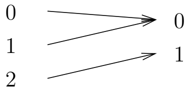
Each sort $\texttt{BV}_{[n]}$ is associated with a domain $D_n = \{0, 1, \ldots, 2^{n-1}\}$
For example $\texttt{BV}_{[4]}$ is associated with $D_4 = \{0, 1, \ldots, 15\}$

As usual, the semantic for the other terms depends on a particular **assignment** to the variables

Each variable $x_{[n]}$ is associated with a function $[\![x_{[n]}]\!]$ of type $D_n \rightarrow \{0, 1\}$

$[\![x_{[3]}]\!]$: $\boxed{1}\boxed{0}\boxed{0}$

$[\![x_{[3]}]\!]$:



$nat_3([\![x_{[3]}]\!])$: 4

$nat_n(\_)$ is a helper meta-function, to facilitate the presentation

Bit-Vector semantic

$$\llbracket c_{[n]} \rrbracket \quad := \quad \lambda x \in [0, n-1].\begin{cases} 0 & \text{if the } x\text{-th bit is } 0 \\ 1 & \text{otherwise} \end{cases}$$

$$\llbracket t_{[l]} :: s_{[k]} \rrbracket \quad := \quad \lambda x \in [0, \ldots, l+k-1].\begin{cases} \llbracket s_{[n]} \rrbracket(x) & \text{if } x < l \\ \llbracket t_{[n]} \rrbracket(x-l) & \text{otherwise} \end{cases}$$

$$\llbracket t_{[n]}[i:j] \rrbracket \quad := \quad \lambda x \in [0, i-j+1].\, \llbracket t_{[n]} \rrbracket(x+j)$$

$$\llbracket t_{[n]} + s_{[n]} \rrbracket \quad := \quad nat_n^{-1}(nat_n(\llbracket t_{[n]} \rrbracket) + nat_n(\llbracket s_{[n]} \rrbracket)) \ \% \ 2^n$$

$$\llbracket t_{[n]} \ \mathbf{AND} \ s_{[n]} \rrbracket \quad := \quad \lambda x \in [0, n-1].\begin{cases} 0 & \text{if } \llbracket t_{[n]} \rrbracket(x) = 0 \\ 0 & \text{if } \llbracket s_{[n]} \rrbracket(x) = 0 \\ 1 & \text{otherwise} \end{cases}$$

$$\llbracket t_{[n]} <_u s_{[n]} \rrbracket \quad := \quad \begin{cases} \top & \text{if } nat_n(\llbracket t_{[n]} \rrbracket) <_u nat_n(\llbracket s_{[n]} \rrbracket) \\ \bot & \text{otherwise} \end{cases}$$

Example 1: C code

Pseudo-code

```
i := 1
while ( i > 0 )
  i := i + 1
```

Example 1: C code

Pseudo-code

```
i := 1
while ( i > 0 )
  i := i + 1
```

C equivalent

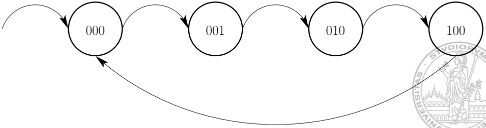
```
unsigned i = 1;
while ( i > 0 )
  i = i + 1;
```

Example 2: C code

Evaluation of $\texttt{BV}_{[32]}$ with C

```
unsigned a = 0xFFFF0000;
unsigned b = 0x0000FFFF;

printf( "a + b   : %8X\n", a + b );
printf( "a * b   : %8X\n", a * b );
printf( "a AND b : %8X\n", a & b );
printf( "a OR b  : %8X\n", a | b );
```

```
module counter(clk, count);

  input clk ;
  output [2:0] count ;
  wire cin ;
  reg [2:0] count ;

  assign cin = ~count [0] & ~count [1] & ~count [2];

  initial begin
    count = 3'b0;
  end

  always @ ( posedge clk )
  begin
    count [0] <= cin;
    count [1] <= count [0];
    count [2] <= count [1];
  end

end module
```



Outline

1. Bit-Vectors
   - Syntax
   - Semantic
   - Examples

2. Solving Bit-Vectors
   - Bit-Blasting
   - Simplifications

Bit-Blasting

Our goal is to devise an automatic decision procedure (SMT-solver) to check the satisfiability of a given Bit-Vector formula

Bit-Blasting

Our goal is to devise an automatic decision procedure (SMT-solver) to check the satisfiability of a given Bit-Vector formula

State-of-the-art techniques are based on reduction to SAT. It is called **bit-blasting**

Bit-Blasting

Our goal is to devise an automatic decision procedure (SMT-solver) to check the satisfiability of a given Bit-Vector formula

State-of-the-art techniques are based on reduction to SAT. It is called **bit-blasting**

- the formula is seen as a circuit, in which variables and constants are inputs, while other terms are intermediate nodes. The outermost Boolean connective or predicate represents the output

Bit-Blasting

Our goal is to devise an automatic decision procedure (SMT-solver) to check the satisfiability of a given Bit-Vector formula

State-of-the-art techniques are based on reduction to SAT. It is called **bit-blasting**

- the formula is seen as a circuit, in which variables and constants are inputs, while other terms are intermediate nodes. The outermost Boolean connective or predicate represents the output
- each variable is assigned to a vector of Boolean variables (n variables for a variable of sort $\texttt{BV}_{[n]}$)

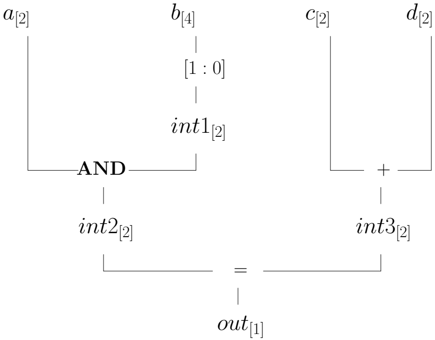
Bit-Blasting

Our goal is to devise an automatic decision procedure (SMT-solver) to check the satisfiability of a given Bit-Vector formula

State-of-the-art techniques are based on reduction to SAT. It is called **bit-blasting**

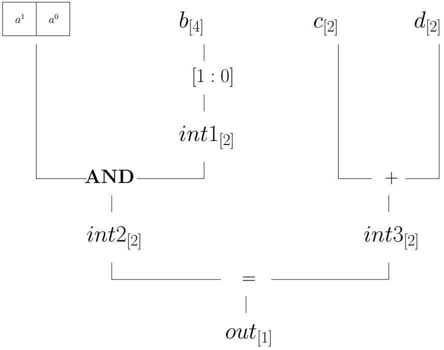
- the formula is seen as a circuit, in which variables and constants are inputs, while other terms are intermediate nodes. The outermost Boolean connective or predicate represents the output
- each variable is assigned to a vector of Boolean variables (n variables for a variable of sort $\mathtt{BV}_{[n]}$)
- each intermediate node is assigned to a vector of Boolean formulæ (n formulæ for a term of sort $\mathtt{BV}_{[n]}$)

$$(a_{[2]}\ \mathbf{AND}\ b_{[4]}[1:0]) = (c_{[2]} + d_{[2]})$$



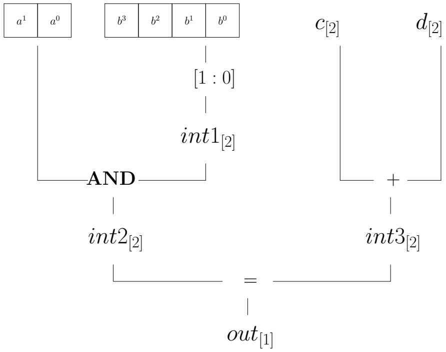
Bit-Blasting

$$(a_{[2]}\ \mathbf{AND}\ b_{[4]}[1:0]) = (c_{[2]} + d_{[2]})$$

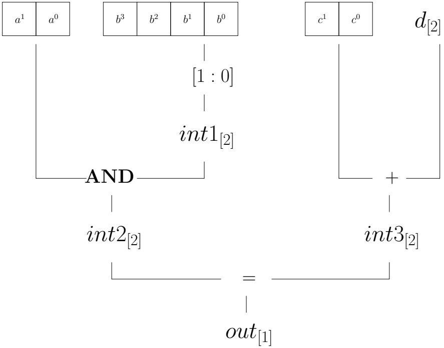


Bit-Blasting

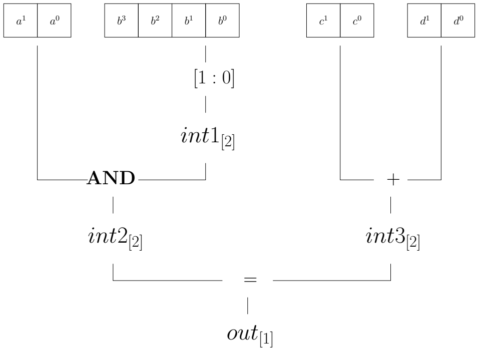
$$(a_{[2]}\ \mathbf{AND}\ b_{[4]}[1:0]) = (c_{[2]} + d_{[2]})$$



$$(a_{[2]} \ \mathbf{AND} \ b_{[4]}[1:0]) = (c_{[2]} + d_{[2]})$$

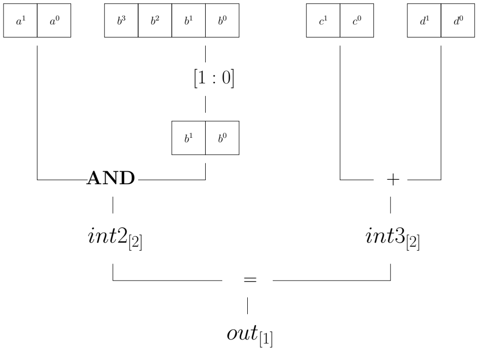


$$(a_{[2]}\ \mathbf{AND}\ b_{[4]}[1:0]) = (c_{[2]} + d_{[2]})$$



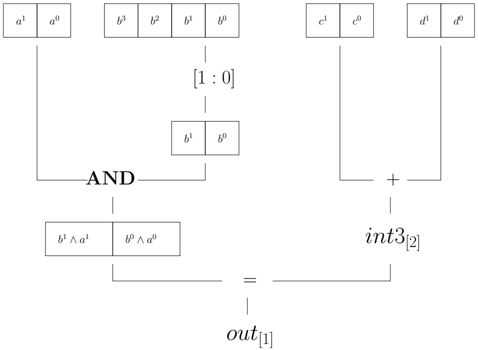
Bit-Blasting

$$(a_{[2]}\ \mathbf{AND}\ b_{[4]}[1:0]) = (c_{[2]} + d_{[2]})$$



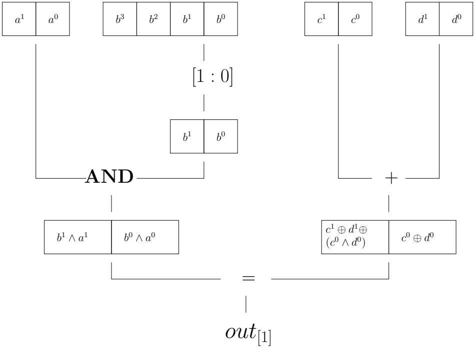
Bit-Blasting

$$(a_{[2]}\ \mathbf{AND}\ b_{[4]}[1:0]) = (c_{[2]} + d_{[2]})$$

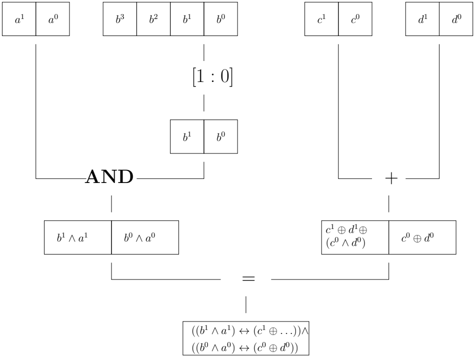


Bit-Blasting

$$(a_{[2]} \textbf{ AND } b_{[4]}[1:0]) = (c_{[2]} + d_{[2]})$$

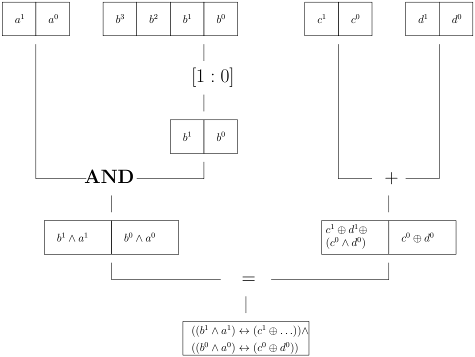


$$(a_{[2]}\ \mathbf{AND}\ b_{[4]}[1:0]) = (c_{[2]} + d_{[2]})$$



Bit-Blasting

$$(a_{[2]}\ \mathbf{AND}\ b_{[4]}[1:0]) = (c_{[2]} + d_{[2]})$$



$$((b^1 \wedge a^1) \leftrightarrow (c^1 \oplus d^1 \oplus (c^0 \wedge d^0))) \wedge ((b^0 \wedge a^0) \leftrightarrow (c^0 \oplus d^0))$$

Bit-Blasing Algorithm (1)

BB := {}, C := {}

Procedure Bit-Blast-Term(t : $\texttt{BV}_{[n]}$ term)

if ($t \in$ C) **return**; // If already in cache, skip
else C := C $\cup$ t // Put in cache

Bit-Blasing Algorithm (1)

BB := {}, C := {}

Procedure Bit-Blast-Term(t : $\texttt{BV}_{[n]}$ term)

if ($t \in$ C) **return**; // If already in cache, skip
else C := C $\cup$ t // Put in cache

if (t is a $\texttt{BV}_{[n]}$ variable)
// Let x be the name of the variable
BB := BB $\cup$ $\{x \mapsto [x^{n-1}, \ldots, x^0]\}$
// where x^i are fresh Boolean variables

BB := {}, C := {}

Procedure Bit-Blast-Term(t : $\mathtt{BV}_{[n]}$ term)

if ($t \in$ C) **return**; // If already in cache, skip
else C := C $\cup$ t // Put in cache

if (t is a $\mathtt{BV}_{[n]}$ variable)
// Let x be the name of the variable
BB := BB $\cup$ $\{x \mapsto [x^{n-1}, \ldots, x^0]\}$
// where x^i are fresh Boolean variables

else if (t is a $\mathtt{BV}_{[n]}$ constant)
// Let c be the constant
BB := BB $\cup$ $\{c \mapsto [c^{n-1}, \ldots, c^0]\}$
// where c^i is $\perp$ if the i-th bit of c is 0, $\top$ otherwise

Bit-Blasing Algorithm (1)

BB := {}, C := {}

Procedure Bit-Blast-Term(t : $\texttt{BV}_{[n]}$ term)

if ($t \in$ C) **return**; // If already in cache, skip
else C := C $\cup$ t // Put in cache

if (t is a $\texttt{BV}_{[n]}$ variable)
// Let x be the name of the variable
BB := BB $\cup$ $\{x \mapsto [x^{n-1}, \ldots, x^0]\}$
// where x^i are fresh Boolean variables

else if (t is a $\texttt{BV}_{[n]}$ constant)
// Let c be the constant
BB := BB $\cup$ $\{c \mapsto [c^{n-1}, \ldots, c^0]\}$
// where c^i is $\perp$ if the i-th bit of c is 0, $\top$ otherwise

else if (t is (t_1 **AND** t_2), and t_1, t_2 are $\texttt{BV}_{[n]}$ terms)
Bit-Blast-Term(t_1)
Bit-Blast-Term(t_2)
BB := BB $\cup$ $\{t \mapsto [\text{BB}(t_1, n-1) \wedge \text{BB}(t_2, n-1), \ldots, \text{BB}(t_1, 0) \wedge \text{BB}(t_2, 0)]\}$
...

where BB(t, i) means:

1. retrieve the correspondence $t \mapsto [t^n - 1, \ldots, t^0]$, and
2. return t^i

Bit-Blasing Algorithm (2)

Procedure Bit-Blast(φ : $\texttt{BV}_{[n]}$ formula)

if (φ is $(t_1 = t_2)$, and t_1, t_2 are $\texttt{BV}_{[n]}$ terms)
    Bit-Blast-Term(t_1)
    Bit-Blast-Term(t_2)
    BB := BB $\cup$ $\{\varphi \mapsto ((\text{BB}(t_1, n-1) \leftrightarrow \text{BB}(t_2, n-1)) \wedge \ldots \wedge (\text{BB}(t_1, 0) \leftrightarrow \text{BB}(t_2, 0)))\}$

Bit-Blasing Algorithm (2)

Procedure Bit-Blast(φ : $\texttt{BV}_{[n]}$ formula)

if (φ is $(t_1 = t_2)$, and t_1, t_2 are $\texttt{BV}_{[n]}$ terms)
    Bit-Blast-Term(t_1)
    Bit-Blast-Term(t_2)
    BB := BB $\cup$ $\{\varphi \mapsto ((\mathrm{BB}(t_1, n-1) \leftrightarrow \mathrm{BB}(t_2, n-1)) \wedge \ldots \wedge (\mathrm{BB}(t_1, 0) \leftrightarrow \mathrm{BB}(t_2, 0)))\}$

else if (φ is $(t_1 <_u t_2)$, and t_1, t_2 are $\texttt{BV}_{[n]}$ terms)
    Bit-Blast-Term(t_1)
    Bit-Blast-Term(t_2)
    BB := BB $\cup$ $\{\ldots\}$

Bit-Blasing Algorithm (2)

Procedure Bit-Blast(φ : $\texttt{BV}_{[n]}$ formula)

if (φ is $(t_1 = t_2)$, and t_1, t_2 are $\texttt{BV}_{[n]}$ terms)
  Bit-Blast-Term(t_1)
  Bit-Blast-Term(t_2)
  BB := BB $\cup$ $\{\varphi \mapsto ((\text{BB}(t_1, n-1) \leftrightarrow \text{BB}(t_2, n-1)) \wedge \ldots \wedge (\text{BB}(t_1, 0) \leftrightarrow \text{BB}(t_2, 0)))\}$

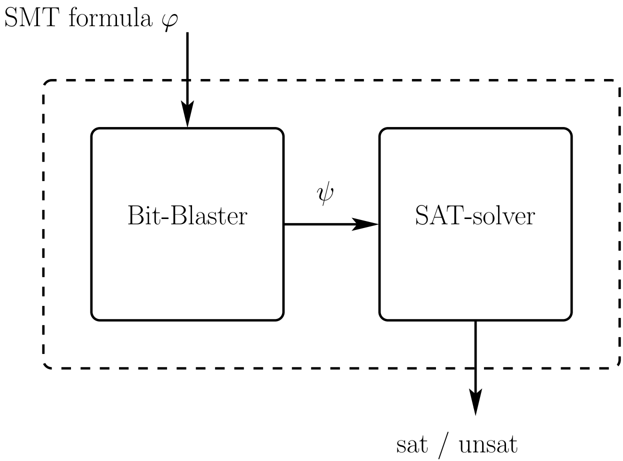
else if (φ is $(t_1 <_u t_2)$, and t_1, t_2 are $\texttt{BV}_{[n]}$ terms)
  Bit-Blast-Term(t_1)
  Bit-Blast-Term(t_2)
  BB := BB $\cup$ $\{\ldots\}$

else if (φ is $\varphi_1 \wedge \varphi_2$ are $\texttt{BV}_{[n]}$ formula)
  Bit-Blast(φ_1)
  Bit-Blast(φ_2)
  BB := BB $\cup$ $\{\varphi \mapsto (\text{BB}(\varphi_1) \wedge \text{BB}(\varphi_2))\}$

...

where $BB(\varphi)$ means:

1. retrieve the correspondence $\varphi \mapsto \psi$, and
2. return ψ


SMT formula φ
Bit-Blaster
ψ
SAT-solver
sat / unsat

Bit-Blasing pros and cons

Pros

- Very easy to write, if compared to write a native Bit-Vector solver

Bit-Blasing pros and cons

Pros

- Very easy to write, if compared to write a native Bit-Vector solver
- Boolean model from SAT can be mapped back to a model for each Bit-Vector variable. If $x_{[n]}$ was bit-blasted as $x^{n-1}, \ldots, x^0$
  - retrieve SAT assignment for each x^i (e.g., $x^0 = \top$, $x^1 = \bot$)
  - construct actual value for $x_{[n]}$ by mapping $\top$ to 1 and $\bot$ to 0 (e.g., $x_{[2]} = 01$)

Bit-Blasing pros and cons

Pros

- Very easy to write, if compared to write a native Bit-Vector solver
- Boolean model from SAT can be mapped back to a model for each Bit-Vector variable. If $x_{[n]}$ was bit-blasted as $x^{n-1}, \ldots, x^0$
  - retrieve SAT assignment for each x^i (e.g., $x^0 = \top$, $x^1 = \bot$)
  - construct actual value for $x_{[n]}$ by mapping $\top$ to 1 and $\bot$ to 0 (e.g., $x_{[2]} = 01$)

Cons

- Does not scale very well. Consider the formula $\neg(x_{[n]} = 0_{[32]}) \wedge (x_{[n]}\,\mathbf{AND}\,y_{[n]}) = (x_{[n]} + y_{[n]})$. It is unsat for **every** n.

Bit-Blasing pros and cons

Pros

- Very easy to write, if compared to write a native Bit-Vector solver

- Boolean model from SAT can be mapped back to a model for each Bit-Vector variable. If $x_{[n]}$ was bit-blasted as $x^{n-1}, \ldots, x^0$
  - retrieve SAT assignment for each x^i (e.g., $x^0 = \top$, $x^1 = \bot$)
  - construct actual value for $x_{[n]}$ by mapping $\top$ to 1 and $\bot$ to 0 (e.g., $x_{[2]} = 01$)

Cons

- Does not scale very well. Consider the formula $\neg(x_{[n]} = 0_{[32]}) \wedge (x_{[n]} \,\mathbf{AND}\, y_{[n]}) = (x_{[n]} + y_{[n]})$. It is unsat for **every** n. But to prove it for $n = 32$ requires 64 Boolean variables, to prove it with $n = 1024$ requires 2048 Boolean variables

Bit-Blasing pros and cons

Pros

- Very easy to write, if compared to write a native Bit-Vector solver

- Boolean model from SAT can be mapped back to a model for each Bit-Vector variable. If $x_{[n]}$ was bit-blasted as $x^{n-1}, \ldots, x^0$
  - retrieve SAT assignment for each x^i (e.g., $x^0 = \top$, $x^1 = \bot$)
  - construct actual value for $x_{[n]}$ by mapping $\top$ to 1 and $\bot$ to 0 (e.g., $x_{[2]} = 01$)

Cons

- Does not scale very well. Consider the formula $\neg(x_{[n]} = 0_{[32]}) \wedge (x_{[n]}\, \mathbf{AND}\, y_{[n]}) = (x_{[n]} + y_{[n]})$. It is unsat for **every** n. But to prove it for $n = 32$ requires 64 Boolean variables, to prove it with $n = 1024$ requires 2048 Boolean variables

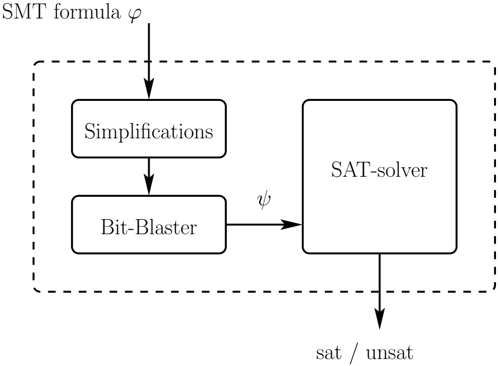
- It destroys the structure of the formula. In the encoding $x_{[32]}$ is not seen as a "single object" but each x^i is unrelated and independent

Simplifications

We use **simplification** rules to fight the two problems in previous slide, before bit-blasting everything

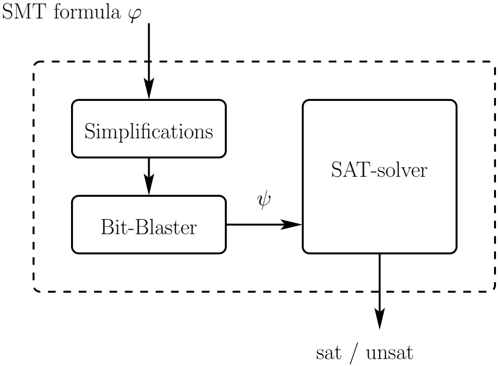
Simplifications

We use **simplification** rules to fight the two problems in previous slide, before bit-blasting everything



Simplifications

We use **simplification** rules to fight the two problems in previous slide, before bit-blasting everything



Simplifications exploit properties of Bit-Vectors to try to reduce the complexity of the formula. We see here some examples, but many more rules do exist. Also, it is very important the way they are combined together

Trivial Simplifications

The following are trivial consequences of the semantic of Bit-Vectors and formulæ in general

Trivial Simplifications

The following are trivial consequences of the semantic of Bit-Vectors and formulæ in general

Bit-Vectors trivial simplifications

- $t = t \ \Rightarrow \ \top$ for a generic term
- $c = d \ \Rightarrow \ \bot$ for two different constants c and d
- $t \,\mathbf{AND}\, 0 \ldots 0 \ \Rightarrow \ 0 \ldots 0$ for a generic term
- $\ldots$

Trivial Simplifications

The following are trivial consequences of the semantic of Bit-Vectors and formulæ in general

Bit-Vectors trivial simplifications

- $t = t \;\Rightarrow\; \top$ for a generic term
- $c = d \;\Rightarrow\; \bot$ for two different constants c and d
- $t \,\mathbf{AND}\, 0\ldots0 \;\Rightarrow\; 0\ldots0$ for a generic term
- $\ldots$

- $\varphi \wedge \varphi \;\Rightarrow\; \varphi$ for a generic formula
- $\varphi \wedge \top \;\Rightarrow\; \varphi$ for a generic formula
- $\varphi \vee \top \;\Rightarrow\; \top$ for a generic formula
- $\ldots$

Ground Term evaluation

If a term is **ground**, i.e., it contains no variables, then it can be always simplified to a single constant

Ground Term evaluation

If a term is **ground**, i.e., it contains no variables, then it can be always simplified to a single constant

Examples:

- $0000 :: 1000 \;\Rightarrow 00001000$
- $0010[1:0] \;\Rightarrow 10$
- $0100 + 0101 \;\Rightarrow 1001$

Variable Elimination Rule

Suppose that the input formula φ is of the kind

$$\varphi' \ \wedge \ (x_{[n]} = t_{[n]})$$

where $x_{[n]}$ is a variable, and $t_{[n]}$ is a term **not containing** $x_{[n]}$

Variable Elimination Rule

Suppose that the input formula φ is of the kind

$$\varphi' \ \wedge \ (x_{[n]} = t_{[n]})$$

where $x_{[n]}$ is a variable, and $t_{[n]}$ is a term **not containing** $x_{[n]}$

then we can rewrite φ as

$$\varphi'[t_{[n]}/x_{[n]}]$$

i.e., we replace every occurrence of $x_{[n]}$ by $t_{[n]}$.

Variable Elimination Rule

Suppose that the input formula φ is of the kind

$$\varphi' \ \wedge \ (x_{[n]} = t_{[n]})$$

where $x_{[n]}$ is a variable, and $t_{[n]}$ is a term **not containing** $x_{[n]}$

then we can rewrite φ as

$$\varphi'[t_{[n]}/x_{[n]}]$$

i.e., we replace every occurrence of $x_{[n]}$ by $t_{[n]}$. We save n Boolean variables in the reduction to SAT

Concatenation Elimination Rule

Suppose that we have the equality

$$t_{[n]} :: s_{[m]} = r_{[n]} :: u_{[m]}$$

then, because the concatenations **match**, we can rewrite it as

Concatenation Elimination Rule

Suppose that we have the equality

$$t_{[n]} :: s_{[m]} = r_{[n]} :: u_{[m]}$$

then, because the concatenations **match**, we can rewrite it as

$$(t_{[n]} = r_{[n]}) \wedge (s_{[m]} = u_{[m]})$$

this rewriting may give more opportunity for applications of previous rules

Exercizes

1. Complete the missing cases in procedures Bit-Blast-Term and Bit-Blast

2. Bit-Blast the formula $\neg(x_{[3]} = 000) \wedge (x_{[3]}\, \mathbf{AND}\, y_{[3]}) = (x_{[3]} + y_{[3]})$

3. Simplify the formula $(x_{[4]} :: y_{[4]}) = (z_{[4]} :: x_{[4]}) \wedge \neg(y_{[4]} = z_{[4]})$