Satisfiability Checking

09 Eager SMT solving for finite-precision bit-vector arithmetic

Prof. Dr. Erika Ábrahám

RWTH Aachen University
Informatik 2
LuFG Theory of Hybrid Systems

WS 22/23

Slides...

...are based on the slides from the Decision Procedures book website.

Motivation

To verify system-level software, we need bit-vector arithmetic - with precise bit-wise operators including e.g. aritmetic overflow.

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

The idea of "bit blasting"

$$
\begin{tikzcd}
\text{bit-vector arithmetic formula} \arrow[d, "\text{Satisfiability-eqivalent transformation}"] \\
\text{propositional logic formula} \arrow[d, "\text{Tseitin's encoding}"] \\
\text{propositional logic formula in CNF} \arrow[d, "\text{SAT solving}"] \\
\text{satisfiability of the propositional logic formula} \arrow[d, "\equiv"] \\
\text{satisfiability of the bit-vector arithmetic formula}
\end{tikzcd}
$$

Finite-precision bit-vector arithmetic: Syntax

Abstract grammar:

$$
\begin{array}{lcl}
\texttt{formula} & ::= & \texttt{formula} \vee \texttt{formula} \;\mid\; \neg\texttt{formula} \;\mid\; \texttt{atom} \\
\texttt{atom} & ::= & \texttt{boolId} \;\mid\; \texttt{term}[\texttt{constant}] \;\mid\; \texttt{term rel term} \\
\texttt{rel} & ::= & = \;\mid\; < \\
\texttt{term} & ::= & \texttt{constant} \;\mid\; \texttt{theoryId} \;\mid\; \sim\texttt{term} \;\mid \\
 & & \texttt{term op term} \;\mid\; \texttt{atom?term:term} \;\mid \\
 & & \texttt{term}[\texttt{constant:constant}] \;\mid\; \texttt{ext(term)} \\
\texttt{op} & ::= & + \;\mid\; - \;\mid\; \cdot \;\mid\; / \;\mid \\
 & & << \;\mid\; >> \;\mid\; \& \;\mid\; | \;\mid\; \oplus \;\mid\; \circ
\end{array}
$$

$\sim x$: bit-wise negation of x

$x << d$: left-shift with distance d

$ext(x)$: sign- or zero-extension of x

$x \circ y$: concatenation of x and y

连接x和y

Bit-vectors

Definition (Bit-vector)

A **bit-vector** x of length ℓ (also written $x_{[\ell]}$) is a function

$$x : \{0, \ldots, \ell - 1\} \to \{0, 1\} \, .$$

We also write x_i for $x(i)$, and use the graphical illustration:

$x_{\ell-1}$	$x_{\ell-2}$	$\cdots$	x_2	x_1	x_0

Semantics of bitvector expressions

The semantics $[\![\cdot]\!]$ of bitvectors depends on the length ℓ of the bit-vectors and the meaning of their bits, specified by an encoding.

Notation: we write $x_{[\ell]U}$ resp. $x_{[\ell]S}$ to annotate a bitvector with its intended encoding.

Binary encoding: $[\![x_{[\ell]U}]\!] := \sum_{i=0}^{\ell-1} x_i \cdot 2^i$

Two's complement: $[\![x_{[\ell]S}]\!] := -2^{\ell-1} \cdot x_{\ell-1} + \sum_{i=0}^{\ell-2} x_i \cdot 2^i$

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

Examples:

$$
\begin{aligned}
[\![11001000_{[8]U}]\!] &= 128 + 64 + 8 = 200 \\
[\![11001000_{[8]S}]\!] &= -128 + 64 + 8 = -56 \\
[\![01100100_{[8]S}]\!] &= 100
\end{aligned}
$$

Semantics of arithmetic expressions

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!

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

$\Rightarrow$ Bit-vector arithmetic uses modulo computations!

Semantics for arithmetic expressions and constraints

Semantics for addition and subtraction (we omit mixed encodings):

$$
\begin{array}{lllcllllll}
\llbracket a_{[\ell]U} & +_{[\ell]U} & b_{[\ell]U} \rrbracket & = & (\llbracket a_{[\ell]U} \rrbracket & + & \llbracket b_{[\ell]U} \rrbracket) & \bmod & 2^{\ell} \\
\llbracket a_{[\ell]U} & -_{[\ell]U} & b_{[\ell]U} \rrbracket & = & (\llbracket a_{[\ell]U} \rrbracket & - & \llbracket b_{[\ell]U} \rrbracket) & \bmod & 2^{\ell} \\[1em]
\llbracket a_{[\ell]S} & +_{[\ell]S} & b_{[\ell]S} \rrbracket & = & (\llbracket a_{[\ell]S} \rrbracket & + & \llbracket b_{[\ell]S} \rrbracket & \bmod & 2^{\ell} \\
\llbracket a_{[\ell]S} & -_{[\ell]S} & b_{[\ell]S} \rrbracket & = & (\llbracket a_{[\ell]S} \rrbracket & - & \llbracket b_{[\ell]S} \rrbracket) & \bmod & 2^{\ell}
\end{array}
$$

Semantics for $<$:

$$
\begin{array}{lllclll}
\llbracket a_{[\ell]U} & < & b_{[\ell]U} \rrbracket & = \mathit{true} & \iff & \llbracket a_{[\ell]U} \rrbracket & < & \llbracket b_{[\ell]U} \rrbracket \\
\llbracket a_{[\ell]S} & < & b_{[\ell]S} \rrbracket & = \mathit{true} & \iff & \llbracket a_{[\ell]S} \rrbracket & < & \llbracket b_{[\ell]S} \rrbracket
\end{array}
$$

Other arithmetic functions and predicates are similar and not detailed here.

Semantics of logical bit-wise operators

We use λ-expressions to give semantics to the logical bit-wise operators.

Examples:

- The zero bit-vector of length ℓ:

$$\lambda i \in \{0, \ldots, \ell - 1\}.\ 0$$

- The function inverting (flipping) all bits of a bitvector of length ℓ:

$$bv\_invert \ := \ \lambda x.\ \lambda i \in \{0, \ldots, \ell - 1\}.\ \neg x_i$$

- The function of bit-vise or for two bit-vectors of length ℓ:

钳住

$$bv\_or \ := \ \lambda x.\ \lambda y.\ \lambda i \in \{0, \ldots, \ell - 1\}.\ x_i \vee y_i$$

The semantics of the other bit-wise operators is defined analogously.

The semantics of Boolean connectors $\wedge, \vee, \ldots$ is as in propositional logic.

Example

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

$$(\lambda i \in \{0, \ldots, 14\}.\ (i < 5)?y_i : x_{i-5})\ [14] \iff x_9$$

$$x_9 \iff x_9$$

$$\mathit{true}$$

Complexity

- The satisfiability problem for bit-vector arithmetic is undecidable for an unbounded width, even without arithmetic.

- It is NP-complete otherwise.

A simple decision procedure for satisfiability

- The most commonly used decision procedure is called bit-blasting.
- It transforms bit-vector arithmetic formulas to satisfiability-equivalent propositional logic formulas.

Definition (Eager satisfiability modulo bit-vector arithmetic solving)

replace each bit vector arithmetic term by variable

1. Build the propositional flattening (Boolean skeletton) as before.
2. Add a Boolean variable for each bit of each sub-expression (term).
3. Add constraints to define the meaning of each sub-expression.

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

i-th bit of term t

What constraints do we generate for a given term?

Easy for logical bit-wise operators.

E.g. for a sub-expression $a \mid_{[\ell]} b$ with new Boolean variables $\mu(a \mid_{[\ell]} b)_i = c_i$, $i = 0, \ldots, \ell - 1$ we add:

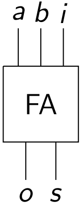
$$\bigwedge_{i=0}^{\ell-1} (c_i \Leftrightarrow (a_i \vee b_i))$$

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

What constraints do we generate for arithmetic terms?

What constraints do we add for $a + b$ where a and b are bits?

$\longrightarrow$ We can build a circuit that adds them!



Full adder:

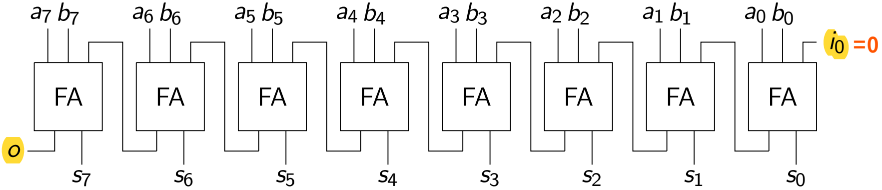
$$\begin{aligned}
o &\equiv (a+b+i)\ \mathit{div}\ 2 &&\equiv (a \wedge b) \vee (a \wedge i) \vee (b \wedge i) \\
s &\equiv (a+b+i)\ \mathit{mod}\ 2 &&\equiv a \oplus b \oplus i
\end{aligned}$$

$$\begin{aligned}
o :\ & (a \vee b \vee \neg o) \wedge (a \vee \neg b \vee i \vee \neg o) \wedge (a \vee \neg b \vee \neg i \vee o) \wedge \\
& (\neg a \vee b \vee i \vee \neg o) \wedge (\neg a \vee b \vee \neg i \vee o) \wedge (\neg a \vee \neg b \vee o) \\
s :\ & (a \vee b \vee i \vee \neg s) \wedge (a \vee b \vee \neg i \vee s) \wedge (a \vee \neg b \vee i \vee s) \wedge \\
& (a \vee \neg b \vee \neg i \vee \neg s) \wedge (\neg a \vee b \vee i \vee s) \wedge (\neg a \vee b \vee \neg i \vee \neg s) \wedge \\
& (\neg a \vee \neg b \vee i \vee \neg s) \wedge (\neg a \vee \neg b \vee \neg i \vee s)
\end{aligned}$$

Number of clauses: $6 + 8 = 14$

What constraints do we generate for arithmetic terms?

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



- Also called carry chain adder
- Adds 2ℓ variables
- Adds 14ℓ clauses

Multiplication

- 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

- Counterexample-guided abstraction refinement (CEGAR) idea: start with the Boolean skeletton and add constraints incrementally only "when needed"

Learning target

- How can we build (finite precision) bit-vector arithmetic formulas?
- What is the meaning of these formulas?
- How can we transform (finite precision) bit-vector arithmetic formulas to satisfiability-equivalent propositional logic formulas?