15-414: Bug Catching: Automated Program Verification

Lecture Notes on Bit-Blasting

Matt Fredrikson

Carnegie Mellon University
Lecture 16
Tuesday, March 22, 2022

1 Introduction

In the previous lectures, we have introduced decision procedures for propositional logic, and saw how to encode values from finite domains into propositional logic formulas. In this lecture, we will expand on the idea of encoding finite domains as propositional formulas, and show how to encode the semantics of arithmetic and bitwise operations over machine integers. This technique, colloquially known as *bit-blasting*, is implemented by nearly all general-purpose SMT solvers, as well as in numerous verification tools. In the next lecture, we will see how bit-blasting allows us to reduce verification problems like those that were the focus of earlier lectures, to propositional satisfiability.

Learning Goals.

In this lecture, you will learn:

- The theory of finite-width bit-vector arithmetic, which we can use to write verification conditions for programs that operate on machine integers.
- Mirroring the use of machine integers in conventional programming languages, the semantics of operations can depend on whether the value is signed or unsigned.
- A procedure for deciding bit-vector formulas by reduction to propositional satisfiability, known as *bit-blasting*.

2 Review: Propositional Encodings

Recall that in the previous lecture we learned how to encode values from finite domains in propositional logic using *unary* and *binary* representations. The intuition behind these representations is that an *unary* representation considers a Boolean variable for each possible value, while a *binary* representation considers the binary representation of an integer.

Suppose we want to encode the domain of an integer variable $\mathcal{X} = \{1, 2, 3\}$. We cannot use a single propositional variable to encode the values that an element from this domain might take, because there are three possible values, and propositional variables can only be $true$ or $false$. We will obviously need to encode elements of this domain using multiple propositional variables, but deciding how many variables and any additional constraints that must be introduced raises questions about which encoding is most appropriate.

Unary representation Consider the auxiliary variables x_1, x_2, x_3. We want to encode the meaning that x_i is *true* iff $X = i$. To encode this property we need to encode that:

1. At least one of these variables must occur:
   $(x_1 \vee x_2 \vee x_3)$
2. At most one of these variables must occur:
   $(\neg x_1 \vee \neg x_2) \wedge (\neg x_1 \vee \neg x_3) \wedge (\neg x_2 \vee \neg x_3)$

Binary representation Consider the binary representation of integers and the auxiliary variables b_1, b_0. We want to encode the following property:

- If $X = 1$ then $b_0 = 0 \wedge b_1 = 0$
- If $X = 2$ then $b_0 = 0 \wedge b_1 = 1$
- If $X = 3$ then $b_0 = 1 \wedge b_1 = 0$

In this case, the meaning of each variable can be used to implicitly encode the possible values of X. The only information we need to encode is possible integer values that are *not part of the domain* of X. In this case, $X = 4$ is not part of the domain but can be encoded using these two variables, therefore we need to disallow this value from occurring by adding the clause $(\neg b_0 \vee \neg b_1)$.

Properties of encodings The binary encoding requires fewer variables, and in many cases, does not require additional constraints to make the encoding faithful to the original finite domain. Because of this, it may at first seem to be the most sensible choice for most settings, but this is not necessarily the case. In fact, the additional constraints imposed by the unary encoding often prove useful for SAT solvers that employ Boolean Constraint Propagation, as they encode relationships between the encoding variables

that can help prune the solver's search space. This is embodied by *consistency* and arc-consistency, defined below.

Definition 1 (Consistent Encoding)**.** An encoding is *consistent* if, when given a partial propositional assignment that is not compatible with any solution to the domain, unit propagation leads to a conflict.

Definition 2 (Arc-Consistent Encoding)**.** An encoding is *arc-consistent* if it is consistent, and additionally unit propagation on a partial assignment discards inconsistent values for the encoding variables.

While the unary encoding enjoys these properties, and its additional constraints can aid a solver in deciding whether a formula is satisfiable, in many cases the cardinality of the domain is simply too large to encode by creating a distinct variable for each possible value. In such cases, we must opt for a binary encoding, which we will later see when we consider how to decide bit-vector formulas via propositional satisfiability.

3 Theory of Bit Vector Arithmetic

Computer systems use *bit vectors* to encode numbers. A bit vector b, as the name suggests, is a finite sequence of binary values of length ℓ. We denote the domain of bit vectors of length ℓ as $\mathbf{BV}_\ell$, and the ith bit of a bit vector b as b_i.

3.1 Syntax and Semantics

Until now, the logical formulas that we've considered have assumed that variables range over the set of mathematical integers $\mathbb{Z}$. When writing expressions involving integer variables, we allowed the usual set of operations: addition, subtraction, multiplication, division, etc. In today's lecture, we will assume that variables refer to bit vectors of a given width ℓ. We can still perform the "normal" set of arithmetic operations on them, but will additionally allow operations like bitwise left and right shift ($\ll, \gg$), bitwise "and" and "or" ($\&, |$), and bitwise exclusive-or ($\oplus$).

$$\begin{array}{rrcl} \text{Bit Vector Expressions} & e & ::= & b \mid x \mid {\sim}e \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 * e_2 \mid e_1 / e_2 \mid \\ & & & e_1 \ll e_2 \mid e_2 \gg e_2 \mid e_1 \,\&\, e_2 \mid e_1 \mid e_2 \mid e_1 \oplus e_2 \end{array}$$

Note that we have so far used the letter c to denote integer constants, and we now use b to distinguish the fact that the constant is a bit vector.

Recall that we defined the meaning, or semantics, of expressions in terms of a state ω that assigns integers to variable symbols. Likewise, we will now assume that a state ω assigns an element from $\mathbf{BV}_\ell$ to variables. Note that for a given variable x, the notation $\omega(x)$ represents a bit vector value, so we may write $\omega(x)_i$ to refer to its ith bit.

When it comes to defining the value of a bit vector expression e under state ω, we must decide whether we intend to interpret bit vector values as *signed* or *unsigned*. We will distinguish these cases with the subscript S (for signed) or U (for unsigned) on the

double-bracket notation for term semantics, i.e., $[\![e]\!]_S$ denotes the value of e as a signed bit vector, and $[\![e]\!]_U$ the value as an unsigned bit vector.

The primary difference between the signed and unsigned cases arises in how we interpret constants. For unsigned bit vectors, we use the normal binary encoding of integers.

$$\omega[\![b]\!]_U = \sum_{i=0}^{\ell-1} b_i 2^i \tag{1}$$

For signed bit vectors, we use the twos-complement encoding, which interprets the bit at position $\ell - 1$ as the *sign bit*:

$$\omega[\![b]\!]_S = -b_{\ell-1} 2^{\ell-1} \sum_{i=0}^{\ell-2} b_i 2^i \tag{2}$$

So for example,

$$\begin{array}{rcl} \omega[\![0101]\!]_S & = & 5 \\ \omega[\![0101]\!]_U & = & 5 \\ \omega[\![1101]\!]_S & = & -3 \\ \omega[\![1101]\!]_U & = & 13 \end{array}$$

Note that the range of integer values assigned to unsigned to unsigned bit vectors is $[0, 2^\ell - 1]$, whereas for signed bit vectors it is $[-2^{\ell-1}, 2^{\ell-1} - 1]$.

Arithmetic operations on bit vectors must account for the fact that $\mathbf{BV}_\ell$ is a finite domain. This amounts to doing modular arithmetic over 2^ℓ, for both signed and unsigned bit vectors.

$$\begin{array}{rcl} \omega[\![e_1 + e_2]\!]_{S,U} & = & \omega[\![e_1]\!]_{S,U} + \omega[\![e_2]\!]_{S,U} \bmod 2^\ell \\ \omega[\![e_1 - e_2]\!]_{S,U} & = & \omega[\![e_1]\!]_{S,U} - \omega[\![e_2]\!]_{S,U} \bmod 2^\ell \\ \omega[\![e_1 * e_2]\!]_{S,U} & = & \omega[\![e_1]\!]_{S,U} * \omega[\![e_2]\!]_{S,U} \bmod 2^\ell \\ \omega[\![e_1 / e_2]\!]_{S,U} & = & \omega[\![e_1]\!]_{S,U} / \omega[\![e_2]\!]_{S,U} \bmod 2^\ell \\ \ldots \end{array}$$

For example, consider taking $5 + 6$ in the signed case when $\ell = 4$. Notice that the result is larger than $2^{\ell-1} - 1 = 7$, which will result in *overflow* and the semantics of the resulting bit vector will be negative:

$$\begin{array}{rcl} \omega[\![0101 + 0110]\!]_S & = & \omega[\![0101]\!]_S + [\![0110]\!]_S \bmod 2^\ell \\ & = & 5 + 6 \bmod 16 \\ & = & 11 \bmod 16 \\ & = & -5 \bmod 16 \\ & = & \omega[\![1011]\!]_S \end{array}$$

The left shift operator works identically for both signed and unsigned bit vectors: when the right operand is d, the least significant $\ell - d$ bits are shifted left d positions, the d most significant bits are truncated, and d zeros are placed in the least significant positions. The right shift operator works differently depending on whether the left operand is signed or unsigned; in the unsigned case, the d most significant bits are

replaced with zeros, whereas in the signed case they are replaced with copies of the sign bit. Note that the right operand for any shift operation is always interpreted as unsigned. In the following, let $d = [\![e_2]\!]_U$:

$$\begin{aligned}
\omega[\![e_1 \ll e_2]\!]_{S,U} &= \omega[\![\underbrace{e_{1,\ell-d} \ldots e_{1,0}}_{\ell-d \text{ bits shifted}} \underbrace{0 \quad \ldots \quad 0}_{d \text{ zeros}}]\!]_{S,U} \\
\omega[\![e_1 \gg e_2]\!]_U &= \omega[\![\underbrace{0 \quad \ldots \quad 0}_{d \text{ zeros}} \underbrace{e_{1,\ell-(d+1)} \cdots e_{1,d}}_{l-d \text{ bits shifted}}]\!]_{S,U} \\
\omega[\![e_1 \gg e_2]\!]_S &= \omega[\![\underbrace{e_{1,\ell-1} \quad \ldots \quad e_{1,\ell-1}}_{d \text{ copies of sign bit}} \underbrace{e_{1,\ell-(d+1)} \cdots e_{1,d}}_{l-d \text{ bits shifted}}]\!]_{S,U}
\end{aligned}$$

The bitwise operators behave as expected: we interpret each bit as a Boolean value, and compute the corresponding logical operation on each pair of corresponding bits.

$$\begin{aligned}
\omega[\![\sim e]\!]_{S,U} &= \omega[\![\neg e_{\ell-1} \ldots \neg e_0]\!]_{S,U} \\
\omega[\![e_1 \,\&\, e_2]\!]_{S,U} &= \omega[\![e_{1,\ell-1} \wedge e_{2,\ell-1} \ldots e_{1,0} \wedge e_{2,0}]\!]_{S,U} \\
&\vdots
\end{aligned}$$

Finally, the relational operators ($\leq$, $=$) make use of the appropriate semantics for signed or unsigned bit vectors, which already map bit vector values to the integers. To specify which semantics to use, we subscript $\models$ with either S or U. So, for example the semantics for inequality of two signed bit vectors is simply,

$$\omega \models_S e_1 \leq e_2 \quad \text{iff } \omega[\![e_1]\!]_S \leq \omega[\![e_1]\!]_S$$

Note, however, that bit vector semantics can lead to surprising results in comparisons if we do not stay alert. The formula $x \leq x + 1$ is valid for mathematical integers, but not bit vectors of any width–signed or unsigned!

4 Bit-Blasting

There are several ways of determining the validity of bit vector formulas. In this section, we will see how to encode a bit vector formula as an equisatisfiable formula in propositional logic. As discussed previously, the validity of a propositional formula can be determined by checking the satisfiability of its negation: if the negation is unsatisfiable, then the formula is valid.

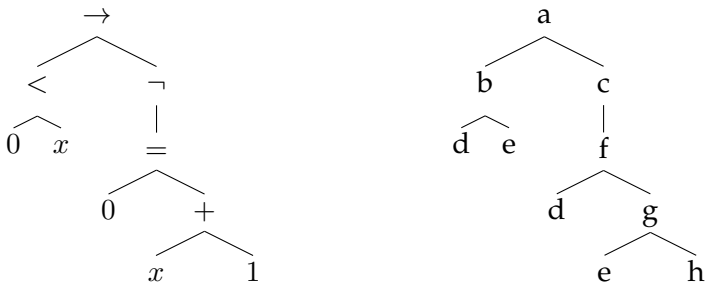
We will see how this is done in steps by showing how to encode the formula in Equation 3.

$$0 < x \rightarrow \neg(x + 1 = 0) \tag{3}$$

Intuitively, we will proceed first in a similar fashion to the Tseitin encoding for producing an equisatisfiable CNF formula, by identifying subformulas and subexpressions. We introduce equivalence constraints for each subexpression, in the manner of the Tseitin encoding, to reflect the Boolean structure of the original formula. For each distinct bit vector expression, we will define a corresponding set of ℓ propositional variables to encode its value. Finally, we introduce constraints over these propositional variables that encode the semantics of bit vector operations.

Step 1: Identify subformulas and subexpressions

As shown in our presentation of the Tseitin encoding, a straightforward way to identify all of the subformulas and expressions of a formula is to represent it as a syntax tree. The formula in Equation 3 would be represented as shown below on the left. For each node of the tree, we create a fresh propositional variable that represents the formula or expression rooted at the node, as shown below on the right.



Note that for identical leaf nodes, we will use the same propositional variable for all instances, as these represent the same variable or constant expression.

Step 2: Encode Boolean structure and bit vector equivalences

Having identified the set of subformulas and expressions, and identified them with fresh variable symbols, we proceed to derive a set of logical equivalences over these fresh variables that encodes the Boolean structure of the original formula, and equalities between bit vector values.

In the running example, this leaves us with the following set of equalities and logical equivalences.

$$\begin{array}{lcl} a & \leftrightarrow & b \rightarrow c \\ b & \leftrightarrow & d < e \\ c & \leftrightarrow & \neg f \\ d & = & 0 \\ e & = & x \\ f & \leftrightarrow & d = g \\ g & = & e + h \\ h & = & 1 \end{array}$$

Note that some of these equalities could be optimized away, reducing the size of the encoding. For example, d could be replaced with the constant 0 wherever it appears, and likewise e could be replaced with x. For the sake of keeping the illustration straightforward, we will leave them as is.

Now that we have identified all of the unique bit vector expressions in the formula, we construct ℓ propositional variables for each variable. Above, a, b, c and f are propositional variables, and d, e, g and h represent bit vectors. We clearly do not need to represent the propositional variables with ℓ *additional* propositional variables, so we would add 4ℓ propositional variables to represent the bit vectors. In the following text,

we will use subscripts on the original bit vector variable name to refer to these propositional variables, i.e., h_0 refers to the variable for the least significant bit of h.

Step 3: Add constraints for bit vector expressions

As this stage, our goal is to replace any equivalence or equality involving a bit vector with a propositional formula that encodes its semantics. The encoding obviously depends on the set of bit vector operators that appear in the formula. We will discuss encodings for relational operators and addition, and refer the reader to Kroening & Strichman [KS08] for details on other operators.

Addition & Subtraction To encode addition, we borrow a page from Boolean circuit construction, and represent an *adder* with propositional logic. A *full adder* is a Boolean circuit that takes as input two bits x, y, along with a carry-in bit c_{in}, and produces their sum z along with the carry-out c_{out}. We encode this functionality with two constraints.

$$\begin{array}{lcl} z & \leftrightarrow & (x \oplus y) \oplus c_{in} \\ c_{out} & \leftrightarrow & (x \wedge y) \vee (c_{in} \wedge x \oplus y) \end{array}$$

The single-bit full-adder can be extended to encode the addition z of two ℓ-bit inputs x, y.

$$\begin{array}{lcl} \neg c_{-1} & & \\ z_i & \leftrightarrow & (x_i \oplus y_i) \oplus c_{i-1} \\ c_i & \leftrightarrow & (x_i \wedge y_i) \vee (c_{i-1} \wedge x_i \oplus y_i) \end{array}$$

Note the constraint $\neg c_{-1}$ to represent that the carry-in bit of the ℓ-bit sum is 0.

Subtraction is encoded using addition, i.e. $x - y = x + (-y)$. The negative of a twos-complement value is obtained by taking its bitwise complement and adding 1, which leads to the following constraints to encode $x - y = z$.

$$\begin{array}{lcl} c_{-1} & & \\ z_i & \leftrightarrow & (x_i \oplus \neg y_i) \oplus c_{i-1} \\ c_i & \leftrightarrow & (x_i \wedge \neg y_i) \vee (c_{i-1} \wedge x_i \oplus \neg y_i) \end{array}$$

Whereas for addition we constrain the initial carry-in bit c_{-1} to be *false* (i.e., 0), for subtraction we assert that it is $true$ to account for the addition of 1 from negating y.

Equality Equality is straightforward to encode, as it amounts to biimplication of all pairs of corresponding bits. When x and y represent bit vectors, we encode as follows.

$$x = y \quad \text{is replaced with} \quad \textstyle\bigwedge_{i=0}^{\ell-1} x_i \leftrightarrow y_i$$

If one side of the equality is a constant, then we just assert the appropriate Boolean constant. For example, the constraints in our running example include $d = 0$ and $h = 1$.

$$\begin{array}{ll} d = 0 & \text{is replaced with} \quad \textstyle\bigwedge_{i=0}^{\ell-1} \neg d_i \\ h = 1 & \text{is replaced with} \quad h_1 \wedge \textstyle\bigwedge_{i=1}^{\ell-1} \neg h_i \end{array}$$

Inequality The encoding for inequality depends on whether the operands are signed or unsigned. In either case, the approach first converts $x < y$ to $x - y < 0$. If the operands are unsigned, then we assert that the carry-out bit of the subtraction is $false$. Note that because we only care about the status of the carry-out bit, we don't need to add constraints for the sum component, as we can compute the carry bits incrementally without keeping track of the sum. We assume that the c_i's are fresh variables in the encoding below.

$$\begin{array}{lcl} c_{-1} & & \\ c_i & \leftrightarrow & (x_i \wedge \neg y_i) \vee (c_{i-1} \wedge x_i \oplus \neg y_i) \\ \neg c_{\ell-1} & & \end{array}$$

If the operands are signed, then we compare the sign bits of the operands with the carry-out bit of their difference. If their sign bits are the same, then the carry-out bit should be $false$; otherwise, it should be $true$. As before, we can ignore the sum bits, and add constraints sufficient to track the carry bits.

$$\begin{array}{lcl} c_{-1} & & \\ c_i & \leftrightarrow & (x_i \wedge \neg y_i) \vee (c_{i-1} \wedge x_i \oplus \neg y_i) \\ c_{\ell-1} & \leftrightarrow & \neg(x_{\ell-1} \leftrightarrow y_{\ell-1}) \end{array}$$

4.1 Running example

We'll conclude this section by illustrating the encoding on the running example, for the simplest case where $\ell = 1$. Recall that we have the following constraints from Step 2.

$$\begin{array}{lcl} a & \leftrightarrow & b \rightarrow c \\ b & \leftrightarrow & d < e \\ c & \leftrightarrow & \neg f \\ d & = & 0 \\ e & = & x \\ f & \leftrightarrow & d = g \\ g & = & e + h \\ h & = & 1 \end{array}$$

For a single-bit width encoding, we introduce five propositional variables to account for the values of the bit vector expressions appearing in the formula: d_0, e_0, g_0, h_0. We then encode the equalities $d = 0$ and $h = 1$:

$$\begin{array}{l} \neg d_0 \\ h_0 \end{array}$$

Next, we account for $e = x$ and $f \leftrightarrow d = g$:

$$\begin{array}{lcl} e_0 & \leftrightarrow & x_0 \\ f & \leftrightarrow & (d_0 \leftrightarrow g_0) \end{array}$$

Next, we encode $g = e + h$:

$$\begin{array}{ll} \neg c_{-1} & \\ g_0 & \leftrightarrow (e_0 \oplus h_0) \oplus c_{-1} \\ c_0 & \leftrightarrow (e_0 \oplus h_0) \vee (c_{-1} \wedge e_0 \oplus h_0) \end{array}$$

Note that the carry-out bit is not used, and the fact that the carry-in bit is $false$ allows us to simplify this to:

$$g_0 \leftrightarrow e_0 \oplus h_0$$

Now we have encoded all of the bit vector expressions and equalities. All that we have left is the inequality $b \leftrightarrow d < e$. As we are encoding one-bit numbers, it doesn't make much sense to assume that their bitvectors are signed twos-complement, so we just assume that they are unsigned. Then we ultimately need to assert that b_0 is true if and only if the carry-out bit of $d - e$ is $false$. Because the variable c is already used in our constraints, we will use v_0 to denote the carry-out of $d - e$.

$$\begin{array}{lcl} v_0 & \leftrightarrow & (d_0 \wedge \neg e_0) \vee (d_0 \oplus \neg e_0) \\ b & \leftrightarrow & \neg v_0 \end{array}$$

We can simplify this by just eliminating v_0:

$$b \quad \leftrightarrow \quad \neg((d_0 \wedge \neg e_0) \vee (d_0 \oplus \neg e_0))$$

This leaves us with the following encoding:

$$\begin{array}{ll} & \neg d_0 \\ \wedge & h_0 \\ \wedge & (e_0 \leftrightarrow x_0) \\ \wedge & (g_0 \leftrightarrow e_0 \oplus h_0) \\ \wedge & (f \leftrightarrow (d_0 \leftrightarrow g_0)) \\ \wedge & b \leftrightarrow \neg((d_0 \wedge \neg e_0) \vee (d_0 \oplus \neg e_0)) \\ \wedge & c \leftrightarrow \neg f \\ \wedge & (a \leftrightarrow b \rightarrow c) \wedge a \end{array}$$

We see that simplifying by propagating $\neg d_0, h_0$ and a gives us the following:

$$\begin{array}{ll} \wedge & f \leftrightarrow x_0 \\ \wedge & b \leftrightarrow x_0 \\ \wedge & c \leftrightarrow \neg f \\ \wedge & b \rightarrow c \end{array}$$

This is equivalent to $\neg x_0$, which is the unsigned bitvector encoding for a satisfying assignment (i.e., 0) to $0 < x \rightarrow \neg(x + 1 = 0)$.

References

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. TACAS'08/ETAPS'08, Berlin, Heidelberg, 2008. Springer-Verlag.

[KS08] Daniel Kroening and Ofer Strichman. *Decision Procedures: An Algorithmic Point of View*. Springer Publishing Company, Incorporated, 1 edition, 2008.