ECE750-28:
Computer-aided Reasoning for Software Engineering
Assignment #1

Due October 4, 2013

October 2, 2013

Problem 1: Warm-up Exercises (25 points)

The following are a set of warm-up problems. Each question is worth 5 points.

1. There are many different Boolean 2-input 1-output functions, such as, AND, OR, NOT, XOR, NAND,... that logicians and engineers have considered. What is the smallest set S of such functions such that one can encode all other 2-input 1-output functions using only the elements of S. (You can assume that the NOT function has two inputs and that they always take the same value. Also, note that we interchangeably use the term 'Boolean function' and 'Boolean connective'.)
2. Can all 2-input 1-output Boolean functions be encoded using only the XOR function?
3. Given n Boolean variables, how many 'semantically' different Boolean functions can you construct?
4. Prove the validity of Demorgan's laws for propositional logic.
5. Explain the difference between equisatisfiability and logical equivalence?

Problem 2: A Basic Reduction (35 points)

Consider a first-order sorted quantifier-free theory of bit-vectors, whose syntax and semantics is described below:

Syntax of well-formed bit-vector formulas

Well-formed formulas in this theory are constructed recursively as defined below. First, we describe how terms in the logic are constructed using constants, variables and functions.

Bit-vector Tems

The atomic terms of the theory are:

- Constants: constants in this theory are finite-length strings over the alphabet $\Sigma = 0, 1$. The letters of Σ are called bits. These constants are called bit-vectors, and the set of constants is represented by C. (Once you are sufficiently familiar with this theory, then the hyphenated word "bit-vector" can be used to refer to both constants and arbitrary terms of the theory.) A n-bit bit-vector constant is an ordered sequences of bits, where the rightmost letter is the 0^{th} bit in the sequence and the leftmost bit is $(n-1)^{th}$ bit in the sequence (and all the letters in between are numbered in the appropriate way).
- Variables: There is an infinite set V of variables represented by letters and numeric subscripts, and they range over these constants. Variables have a length associated with them that specify the length of the bit-vector constants that can be assigned to them.
- Functions: The functions of this theory are concatenation (.), extraction ([i:j]), addition, and bit2bool.

A bit-vector term t of this theory is inductively defined using constants, variables, concatenation (.), extraction ([i:j]), and add functions (+) as follows:

$$t := c \in C \mid v \in V \mid t_1.t_2 \mid t_1[i:j] \mid t_1 + t_2$$

where, t_1, t_2 are bit-vector terms. i, j are natural numbers, where $i >= j$. It is assumed that the length of the variables is always non-negative. It is assumed that the arguments of the + function are always of the same length. The length of the term $t_1.t_2$ is the sum of the length of the terms t_1, t_2. The length of the term $t_1[i:j]$ is $i - j + 1$.

Semantics of Bit-vector Terms

The semantics of logic or theory refers to the meaning we assign to the symbols (connectives, terms and formulas) of the logic or theory. Here we define the semantics of terms:
Constants are interpreted as the corresponding natural numbers in binary encoding. As is typical in binary encodings, the rightmost bit of a constant is the least-significant bit (also called the 0^{th} bit) and the leftmost bit is the most-significant bit (also called the n-1^{th} bit for an n-bit bit-vector). Variables are interpreted over (or given the semantics of) bit-vector constants of the same length. Semantics of concatenation and extraction are as you would expect. The semantics of the add function (denoted as +) are exactly the same as in programming languages (e.g., for 1 bit bit-vector the addition is defined as 0+0 = 0, 0+1=1, 1+0=1, and 1+1=0).
bit2bool takes as input a single bit bit-vector, and returns a Boolean. For example, if the input is the bitvector constant 1 (resp. 0), then the output of bit2bool() is TRUE (resp. FALSE). If the input of bit2bool is non-constant bit-vector term of length 1, then the output is a Boolean variable. The bit2bool is undefined for bit-vectors of length more than 1.

Formulas in the given Bit-vector Theory

Atomic formulas, denoted as F, in this theory are constructed as follows:

$$F := t_1 = t_2 \mid t_1 < t_2$$

where t_1, t_2 are bit-vector terms of the same length. Arbitrary formulas are constructed inductively out of Boolean connectives in the usual way.

Semantics of Formulas

A model (or interpretation) of a formula F in this theory is a set of assignments of bit-vector constants from C to the variables in the terms of F. Variables can only be assigned constants of the same length.
$t_1 = t_2$ is true iff there is an assignment to the variables in t_1, t_2 such that t_1 and t_2 evaluate to exactly the same bit-vector.
$t_1 < t_2$ is true iff there is an assignment A to the variables in t_1, t_2 such that when the bit-vector constants t_1 interpreted as a number is smaller than the bit-vector constant t_2 interpreted as a number under the assignment A.

Reduction

A reduction is a computer program that translates instances of one computational problem into instances of another. To be meaningful, reductions have to be sound, complete, terminating and usually have complexity bounds. The notion of soundness, completeness, termination of reductions are closely related to that of proof systems, but are not the same. We say that a reduction f from problem A to problem B is sound and complete if

$$\forall x \in A \iff f(x) \in B$$

The normal form algorithms that we discussed in class are reductions. We briefly discussed reductions in class, and will go into them in greater detail in future lectures. The point of this question is to help you to develop an intuition for the concept of reduction.

The Question

Describe a reduction from any formula F in the above-described theory to a formula B in Boolean logic such that F is satisfiable iff B is.

Problem 3: Sound and Complete Proof System (40 points)

In class we studied two proof systems, namely, the semantic argument method and the resolution proof system.

What is a Proof System

A proof system for a logic L is typically defined as a finite set of proof rules and axioms. The proof rules are syntactic structures defined over the formulas in L as follows: They typically have formulas called "premises", followed by formulas called "consequents". The interpretation is that the consequents "logically" follow from the premises. Axioms in the proof system are proof rules with no premises, i.e., we know them to be always true (i.e., valid) unconditionally. The consequents, premises and axioms are formulas in L.
A proof in a proof system is a finite directed graph, whose nodes are proof rules or axioms, and a directed edge is drawn from the consequent of a proof rule to the premise of another proof rule if the consequent formula and the premise formula match syntactically (assume that the formulas are in normal form, variables have a total ordering and the variables occuring in a formula are written down according to this total order). The leaves of a such a graph are axioms, and the root is called a theorem. (observe that such graphs look like inverted trees.)

Proof Procedures and SAT Solvers

A proof system can be implemented using a computer program called a PROOF PROCEDURE, where such programs take as input formulas and output "IS A THEOREM" if the input is valid, and "NOT THEOREM" if the input is invalid. Procedures for richer logics may also output "DON'T KNOW" or "TIME OUT".
Given the duality between satisfiability and validity, and their connection to provability, a SAT SOLVER can be converted into a PROOF PROCEDURE.

What is Soundness and Completeness of a Proof System

We say that a proof system (or proof procedure) P is sound if the following is true: For any input formula I, If P outputs "IS A THEOREM" then I is valid.
Conversely, we say that a proof system (or proof procedure) P is complete if the following is true: For any input formula, if I is valid, then P can generate a proof for it in finite time and output "IS A THEOREM". Observe that completeness of P does not imply that it will terminate for all inputs.

The Question

1. Come up with a sound and complete proof system (axioms and proof rules) for proposition logic?
2. Explain in detail why you believe your proof system is sound and complete?

3. Is your proof system terminating? If yes, explain why. If no, explain why not.

4. What is the connection between satisfiability, validity and provability in propostional logic? (**This sub-question is a bonus question.**)