Assignment 3

CS 704, Spring 2015
Due: April 20 2015

1 Linear temporal logic

Part 1  Consider the following formulas in LTL. For each pair of formulas, answer whether they are equivalent or inequivalent. If equivalent, argue why; if inequivalent, provide an infinite path that is a model of one formula but not the other.

1. $Fp \land Fq$ and $F(p \land q)$
2. $XFq$ and $FXq$
3. $X(p U q)$ and $(p U (Xq)) \lor q$
4. $GF(p \lor q)$ and $(GFp) \lor (GFq)$
5. $GG(p \lor \neg q)$ and $\neg F(\neg p \land q)$
6. $(p U q) U q$ and $p U q$

Part 2  Suppose you are only allowed to use the temporal operators $U$ and $X$. Will you be able to express any LTL formula? Prove your answer.

Part 3  Prove validity of the following formulas. A formula is valid if it holds for any model (in this case, models are infinite paths).

1. $(\neg q U (\neg p \land \neg q)) \Rightarrow \neg Gp$
2. $((G\neg q) \land (X\neg p)) \Rightarrow \neg q U (\neg p \land \neg q)$

Part 4  Suppose we have the two atomic propositions $p$ and $q$ in some Kripke structure $M = (S, \rightarrow, S_0, L)$. Say we want to specify the property that for every state $s$ such that $p \in L(s)$ there is at least one state $s'$ such that $(s, s') \in \rightarrow$ and $q \in L(s')$.

Is it possible to express this property in linear temporal logic? If so, provide such formula; if not, argue why it is inexpressible.
2 Model checking

Consider the following program with the single variable $x$, which is a 3-bit unsigned integer. Overflows are defined as usual: if $x$ is 111 and we increment it by 1, it becomes 000. The * value indicates a non-deterministic choice.

```c
int x = 1;
while (*)
    x = x + 1
x = 0;
```

**Part 1** Define a Kripke structure $M$ that models this program. (Your definition need not be explicit, that is, you may define sets of states or transitions mathematically instead of enumerating all possible states.)

**Part 2**

1. Provide an LTL property that holds if and only if all program executions terminate.
2. Provide an LTL property that holds if and only if $x == 0$ whenever the program terminates.

3 Boolean decision diagrams

**Part 1** Provide a reduced ordered BDD with 3 variables that has maximal size.

**Part 2** Provide an example of a reduced ordered BDD that has significantly different sizes for different variable orderings.

4 Program encodings

Consider the following loop-free program with input variable $x$ and return variable $r$. The set of program variables is $\{x, r, y, z\}$. Assume all variables are mathematical integers (i.e., no overflows/underflows).

```c
int y = x+1;
int z = 2*y;
if (z > 0) z = z - 1;
int r = 3*z;
```
Part 1  Provide a logical formula whose models are in one-to-one correspondence with
the concrete executions of the program (that is, for each execution there is a corresponding
model and vice versa). Argue correctness of your solution.
(This exercise follows our transition relation encodings that we’ve defined for predicate
abstraction and bounded model checking.)

Part 2  Provide a logical formula that is satisfiable if and only if there is a program
execution that starts with a positive value for $x$ and ends with a negative value for $r$.

5 Abstract domains

Consider the following program where all variables are mathematical integers.

\begin{verbatim}
x = 10;
y = 10;

while (x + y > 0)
   x = x - 1;
y = y - 1;
z = x + y;

assert (z == 0)
\end{verbatim}

Part 1  Prove that the assertion always holds (i.e., can never be violated) by giving an
inductive loop invariant that is strong enough to imply the assertion. Show that your
answer is indeed an inductive loop invariant.

Part 2  Suppose you were to use an abstract interpreter with an intervals abstract domain.
Will you be able to prove that the assertion always holds? If yes, give an inductive loop
invariant in the intervals domain; if no, argue why no such invariant exists. Recall that
such an invariant would be of the form $x \in [l_x, u_x], y \in [l_y, u_y], z \in [l_z, u_z]$, where $l_i, u_i$ are
integers, $\infty$, or $-\infty$.

Part 3  Suppose you were to use predicate abstraction to prove this program correct. Give
a sufficient set of predicates that will enable constructing a safe inductive loop invariant.
(You may assume that predicate abstraction operates at the granularity of sequences of
statements, that is, every strongest post computation starts from an abstract state and
computes the next state after taking a sequence of instructions, e.g., $x = 10; y = 10$).

3
6 Abstract interpretation and least fixpoints

This question concerns the correctness and efficiency of algorithms for abstract interpretation of flow-chart programs (as defined in class).

For this question, assume that the nodes of the flow chart are numbered from 1 to \( k \), that each node has at most a constant number of predecessors and a constant number of successors, and that the abstract value for a given node comes from a finite-height domain \( D \), where the height of \( D \) is \( h \). The abstract functional \( f_a \) is thus of type \( D_k \rightarrow D_k \), and \( \text{fix} \ f_a \) can be expressed as

\[
\text{fix} \ f_a = \bigcup_{i=0}^{\infty} f_a^i(\bot_D).
\]

(\( f_a \) is the abstract interpreter we defined in class. Recall that we computed the its least fixpoint by iteratively growing it from \( \bot_D \). Here, \( D_k \) is the abstract domain that defines the reachable states at the \( k \) program locations.)

Further assume that we can perform a join of two \( D \) values in constant time, and that we can also iterate over the successors of a flow-chart node in constant time.

Finally, assume that \( f_a \) has the property that when \( f_a \) is applied to a value \( x \in D_k \), the \( i \)th component of its output (which corresponds to flow-chart node \( i \)) depends only on the slots of \( x \) that correspond to node \( i \)s predecessors in the flow chart. Let’s denote the computation that computes the \( i \)th “part” of \( f_a(x) \) by \( f_{a,i}(x) \), and lets assume that an application \( f_{a,i}(x) \) can be performed in constant time. Lets now consider how an algorithm to determine \( \text{fix} \ f_a \) could be implemented as an iterative algorithm in an imperative programming language, such as C. One way would be to have two arrays of \( D \)'s, say \( \text{val}[1..k] \) and \( \text{temp}[1..k] \), and use the following procedure:

```c
for i = 1 to k do
    val[i] = \bot_D
while changes to any components of val occur do
    for i = 1 to k do
        temp[i] = f_{a,i}(val)
    for i = 1 to k do
        val[i] = temp[i]
```

**Part 1** What is the asymptotic worst-case cost for computing \( \text{fix} \ f_a \) by the above procedure?

**Part 2** The above algorithm is “round-robin iteration”. An alternative approach is to use the so-called “chaotic-iteration” algorithm:
for $i = 1$ to $k$ do
    $val[i] = \bot_D$
worklist = \{i | 1 \leq i \leq k\}
while worklist $\neq \emptyset$ do
    select and remove a node-index $i$ from worklist
    $foo = f_{a,i}(val)$
    if $foo \neq val[i]$ then
        $val[i] = foo$
        for each successor $j$ of node $i$ do
            worklist = worklist $\cup \{j\}$
What is the asymptotic worst-case cost of the chaotic-iteration algorithm?

Part 3 It is clear that the round-robin-iteration algorithm finds $fix f_a$, because it directly implements a climb up the Kleene sequence $[\bot_D, f_a(\bot_D), \ldots, f_a^i(\bot_D), \ldots]$. However, it is not so clear what the chaotic-iteration algorithm is doing: Because single slots of val are treated individually on each iteration of the while loop, the chaotic-iteration algorithm is not implementing a climb up the Kleene sequence. The question then is: Is the chaotic-iteration algorithm a correct way of computing $fix f_a$?

For this part, show that it is a correct way of computing $fix f_a$ by arguing that the following properties hold:

1. The chaotic-iteration algorithm terminates.
2. The chaotic-iteration algorithm finds a fixed point.
3. The chaotic-iteration algorithm finds the least fixed point (namely $fix f_a$).