CSC410

Assignment 3

Due on November 29, 2021 at 11:59pm, worth 12% of the course mark

Assignment Format and Guidelines on Submission

Submit a properly **typed** PDF on Markus. No handwritten assignment will be accepted. Unless otherwise specified, no English words will be marked.

List of files to submit:

- `a3.pdf` which will include the cleanly **typed** solutions to the problems.
- `source.zip` which will include the solution to problem 8 as described in the text of the problem.

This assignment is partially released to mirror the progress in class. The remainder of it will be released on Friday November 19. Ideally, you should be finished with this part by November 19th, to keep your workload light from November 19th until the due date of November 29th. Consider the first release to be roughly about half of your work load for this assignment, and plan accordingly.

Problem 1: Symbolic Testing (20 points)

Consider the following function foo

```
 1 int foo(int x, int y)
 2 {
 3     if (x < y) {
 4         x = -x ;
 5         y = -y;
 6     }
 7     if (x <= y){
 8         x = -x;
 9         y = -y;
10     } else{
11         int z = x;
12         x = y;
13         y = z;
14     }
15     assert(x < y);
16 }
```

(a) Enumerate all the paths (regardless of feasibility) in foo where each path is given as a sequence of line numbers (e.g. 3,4,5,10,11,12,18) .

(b) Are all the paths listed above feasible? Prove the paths are not feasible as such by filling instances of a table in the below format (please check tutorial 5 slide 8 and 10 for the format). At the end of each, write a short English description to sum up the result of the table.

Line No.	Assignment	Path Condition

(c) Of all the feasible paths, would any result in an assertion violation? Prove your claim using symbolic execution. Fill in a table in the above format again and conclude your argument using a short English description.

Line No.	Assignment	Path Condition

LTL Problems

Problem 2 (12 points)

Let us assume we have a system with only one observable component: a colour LED light bulb. This light bulb can be of colours white (w), red (r), green (g), and blue (b) when it is on, or it can be off (o). The system changes state every second, and the status of the light accordingly changes (or remains the same). Note that it is assumed that the light is always exactly one colour.

Translate each of the following English specification for this simple system to an LTL formula.

(a) The light bulb turns red exactly once (but it can remain red for any duration).

(b) The light bulb stays on forever, changes colour every second, and alternates between red and white.

(c) The light bulb stays on forever, alternates between colours red, blue, and white (in that order), staying at each colour for an arbitrarily long (non-zero but finite) amount of time.

(d) The light bulb can only turn white if it has been previously at least once blue, once green, and once red (but not necessarily in that order).

Problem 3 (30 points)

Which one of the following equivalences hold? Give a formal proof for the correct ones and provide a counterexample for the incorrect ones. A counterexample is an infinite path that satisfies one side and not the other. You may not use any of the equivalences from the lecture/book as a boost. You are meant to prove these from scratch whenever they hold.

(a) $\varphi \,\mathtt{U}\, \neg\varphi \equiv true$

(b) $(\Diamond\Box\varphi_1) \wedge (\Diamond\Box\varphi_2) \equiv \Diamond(\Box\varphi_1 \wedge \Box\varphi_2)$

(c) $\Box\Diamond\varphi \implies \Box\Diamond\psi \equiv \Box(\varphi \implies \Diamond\psi)$

(d) $\varphi \,\mathtt{U}(\psi \vee \neg\varphi) \equiv \Box\varphi \implies \Diamond\psi$

(e) $\bigcirc\Diamond\varphi \equiv \Diamond\bigcirc\varphi$

Problem 4 (10 points)

Recall that satisfiability and validity of LTL formulas are defined in the same way as propositional logic formulas. An LTL formula φ is **satisfiable** if and only if there exists a path π that satisfies it ($\exists\pi : \ \pi \models \varphi$). An LTL formula φ is valid if and only if all paths π satisfy it ($\forall\pi : \ \pi \models \varphi$). Note that validity of φ can also be reformulated as the equality $\varphi \equiv true$.

For the formulas below, determine if the formula is satisfiable, unsatisfiable, or valid. Formally justify your answer.

(a) $\Diamond b \implies (a \,\mathtt{U}\, b)$.

(b) $\bigcirc(a \vee \Diamond a) \implies \Diamond a$

CTL Problems

Problem 5 (12 points)

Recall the setup of Problem 2 with the LED light bulbs. We will reuse it for this problem to write a few more properties in CTL.

(a) The light bulb turns green exactly twice (but can remain green for any duration each time).

(b) The light bulb can turn from red to blue (without another colour interrupting) in the future. It can also turn from white to blue. But, it can never turn from blue to red.

(c) There is a future in which the light bulb is never indefinitely stuck on any one colour.

(d) If the light bulb has ever switched from white to blue in the past, then it cannot switch from blue to white in the future.

Problem 6 (16 points)

Let TS be a finite transition system (over AP) without terminal states (i.e. every state has an outgoing transition), and Φ and Ψ be CTL state formulae (over AP). Prove or disprove: $TS \models \exists(\Phi \,\mathsf{U}\, \Psi)$ if and only if $TS' \models \exists\Diamond\Psi$ where TS' is obtained from TS by eliminating all outgoing transitions from states s such that $s \models \Psi \vee \neg\Phi$.

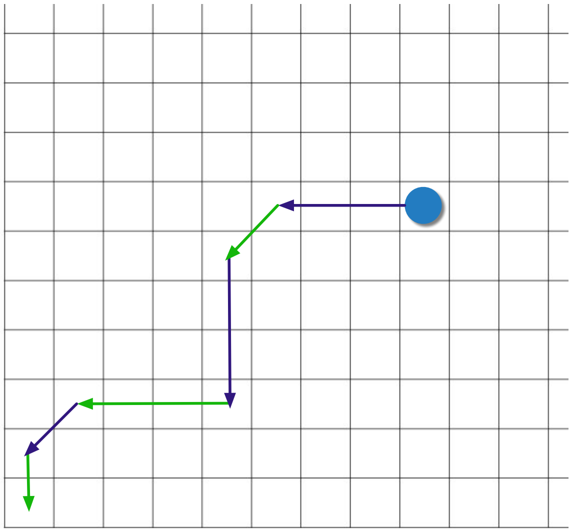
Model Checking

Problem 7 (40 points)

The goal of this problem is to ensure that you understand the ideas behind the CTL model checking algorithm, and specifically the way universal and existential *until* is computed through fixpoints.

There is a game played on a grid of squares with one piece which is initially located at a position (m, n) of a grid (with $m, n \in \mathbb{N}$). The grid's origin $(0, 0)$ is at the bottom left corner and it is arbitrarily large including all squares with pairs of natural number coordinates.

The game is played between two players, who take alternate turns to move this game piece towards the origin. The valid moves for the piece are like a chess queen, as long as the direction of the move is towards the origin, i.e. left, down or diagonally towards left-down. Like a chess queen, the piece is allowed to travel as far as the player chooses in a valid direction during the one move. The player that moves the piece to the origin wins the game. Below is an example play of the game:



played from the initial location $(8, 6)$ where the first player loses the game.

We say a player has a *winning strategy* for a game iff there is play for this player to win this game independent of the choices that the opponent makes. For example, the first player always has a winning strategy from any location $(n, 0)$, $(0, n)$, or (n, n) because the player can move the piece in one move to the origin and win.

A two-player game is called *determined* if from any given position, exactly one of the players has a winning strategy. The above game is determined. Given two excellent players and any location (m, n), either player one always wins the game from (m, n) or he always loses.

The goal of this exercise is to implement a *decision procedure*. The input will be the pair of numbers (m, n). The output is "1" if the first player has a winning strategy from this location, and "2" if the second player has a winning strategy from this location.

Note: this problem is not a random implementation problem. To come up with a solution that scales up, you are encouraged to think carefully about how checking for the existence of a winning strategy relates to the concepts of existential and universal path properties. You are also encouraged to think about the algorithm we discussed for *until* and how the idea behind that algorithm can hint at a nice solution for this problem.

Format

You are free to implement this in the programming language of your choice. Submit your source files as one zipped directory `source.zip`. This directory should include an executable called `game` that runs on the CDF machines. The input is passed to your executable as a command line parameter, that is:

```
./game 2 1
```

should execute on a CDF machine and return "2", since the second player has a winning strategy from the location $(2, 1)$.

Grading

There is a naive algorithm to solve this which will scale very poorly. What does poorly mean? It means that the algorithm has exponential complexity and will likely take over a minute to process a location as small as $(12, 10)$. You may assume that this naive algorithm will get no marks. The reason is that this default naive solution is something anyone who knows programming can implement and has nothing to do with the material taught in this class.

A reasonable algorithm should handle the same location $(12, 10)$ in a small fraction of a second. It would be imprecise to put an exact number on this (since it will be hardware dependent), but think of it as around 0.01s. But, more importantly, you should not see a substantial jumps for small coordinate changes at these values, for example, between the times for $(12, 10)$ and $(13, 12)$. As an another example, think about your algorithm scaling up to around coordinates $(60, 60)$ with the execution time remaining under one minute. A solution like this will take the full mark. But, if you are truly careful with your solution, you should be able to solve any point in the 60×60 grid will be under one second.

You cannot boost your solution by giving it a lookup table for smaller values. For example, one can manually computer all solutions for a 10×10 grid, enter those values as a constant for the algorithm, and let further points to get to one of these points. **This will be considered cheating.**

Note that we will not grade your source code. We ask you to submit it for insurance, that is, in case something goes wrong with the executables and you would like to reclaim your mark through the original material submitted.